




EX LIBRIS
UNIVERSITATIS
ALBERTENSIS

The Bruce Peel
Special Collections
Library



Digitized by the Internet Archive
in 2025 with funding from
University of Alberta Library

<https://archive.org/details/0162015205204>

University of Alberta

Library Release Form

Name of Author: Vadym Voznyuk

Title of Thesis: Real time motion upmapping

Degree: Master of Science

Year this Degree Granted: 2001

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

THE UNIVERSITY OF CHICAGO

LIBRARY

THE UNIVERSITY OF CHICAGO

LIBRARY

THE UNIVERSITY OF CHICAGO


LIBRARY

Animation can explain whatever the mind of man can conceive
Walt Disney

University of Alberta

REAL TIME MOTION UPMAPPING

by

Vadym Voznyuk 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2001

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Real time motion upmapping** submitted by Vadym Voznyuk in partial fulfillment of the requirements for the degree of **Master of Science**.

Abstract

Modern computer games feature various types of animation, in many cases human animation is a key component of the game. Game studios use keyframe animation, motion captured animation or a mix of both to animate the game characters. Motion captured animation is very realistic, but is difficult to control or alter.

The *physically based motion transformation* framework, developed by Popović [78] [79], adds controllability to captured motion sequences. However, the process of the motion transformation is rather complicated, requiring motion preprocessing and postprocessing stages in addition to the motion editing stage. The focus of this thesis is the postprocessing stage of motion transformation (*motion upmapping*), as presented by Popović.

We present the method for real time motion upmapping. The system applies transformed motion to the full character model in real time. Sport simulation computer games are a good application for this technique. In these games the motion is not a linear sequence of events, because the character animation is controlled by the user. Physically based motion transformation in such games can be beneficial in several ways. The transformed motion can be stored in a game in "compressed" form, i.e. as the simplified model motion. During the upmapping stage this simplified motion is restored to the full model motion in real time. By storing the motion in compressed (simplified) form, significant disk space and load time savings can be achieved. Another possibility is a near real time motion retargeting. For example, in a soccer game player can be "injured" by retargeting its motion in the background.

Animators and motion capture studios will benefit from the physically based motion transformation as well. Currently, there are static animation sequences for every particular character motion in the game. In the case of motion captured animation this means running a capture session for every possible motion sequence in the game. With the motion transformation at the end of the motion capture data pipeline, it is now possible to

alter the captured motion in a realistic way. The animators can now adjust the original captured motion, thus saving on the motion capture sessions.

Acknowledgements

First of all, I'd like to thank my supervisor, Dr. John Buchanan. It was he who brought me from the freezing plains of Alberta to warm Vancouver, a city of unbelievably beautiful nature and a burgeoning electronic entertainment industry. As a chief research scientist of the Electronic Arts Canada, Dr. Buchanan was responsible for inspiring and supporting my research. He provided the financial and moral support for my project, as well as being good company on my trip to SIGGRAPH 2000. It was because he was behind my back that I went on the rollerblade ride which ended up in a New Orleans police car. I was sure he could pull me out of anything because he was such a strong person.

Dr. Zoran Popović influenced me a lot and made possible my thesis research. As the author of the source code of the motion transformation system I was working with, he provided me with indispensable assistance in bringing up his system and coping with its complex user interface. In addition, he gave me valuable insights on what it takes to be an academician in a North American university, how research and teaching are bound together and why one might be interested in going academia. I received a good deal of inspiration just by looking at his student project demos and his "personal" mocap and video editing studio.

When you do a thesis, research and development are the first steps along the road, which leads to the dark jungle of written English. For me, as a non-native English speaker, the editing stage was the most important and I could do nothing without serious help from other people. Eric French, a retired high school English teacher, and James Renaud did the majority of the work, rooting out countless grammar errors, fixing missed or misplaced articles and even eradicating a dozen or so typos. Yes, even the spell checkers are puzzled sometimes with my English. Mr. Vincent Manis of Electronic Arts Canada did the second pass, fixing a lot of stylistic mishaps and suggesting valuable additions to several sections in the text. Finally, my faculty supervisor, Dr. Terry Caelli, put a final polish on my work. I'd like to thank them all, since editing and fixing my English is the most unpleasant business I can imagine.

Since life is not only about motion transformation research, I have to mention people who helped me a lot in going through the process of writing a thesis in a new country and a new language. Brian Penny, the chaplain at the University of Alberta is a very special man. He is the keeper of many foreign students, and I had the immense luck to get him as my volunteer guide on Canadian life. The biggest favor he did for me was finding the best apartment in Edmonton where I could stay during my study, but this is only the small part of what he did for me.

My fellow students kept me sane too and created an aura of friendship and goodwill

to help me to adapt to the different lifestyle and language. Pablo Figueroa was good company in our New Orleans SIGGRAPH 2000 trip and very helpful guy in general, while Ehud Sharlin was the local “Counter Strike” and “Half Life” fan. Too bad I had no time to frag him in a CS match. Paul Ferry was one quick talker and a \LaTeX expert. I’d like to thank him for his help in dealing with \LaTeX .

A very special fellow student was Rüyam Acar, whose supply of exotic and sensuous music actually turned by blackest days into something bearable. I dedicate this thesis to her.

Table of Contents

| | | |
|-----------|--|-----------|
| 1 | Introduction | 1 |
| 1.1 | Motion as a research subject | 1 |
| 1.2 | Motion in nature | 2 |
| 1.3 | Motion in computer graphics | 6 |
| 1.4 | Motion in computer games | 6 |
| 1.5 | Motion transformation and computer games | 11 |
| 1.6 | Thesis outline | 13 |
| | | |
| 2 | Background | 14 |
| 2.1 | Overview of the motion synthesis methods | 15 |
| 2.1.1 | Keyframe animation | 15 |
| 2.1.2 | Procedural animation | 18 |
| 2.1.3 | Dynamic simulation | 20 |
| 2.1.3.1 | Forward dynamics | 21 |
| 2.1.3.2 | Optimal control | 23 |
| 2.1.4 | Motion capture | 24 |
| 2.1.5 | Robotics | 26 |
| 2.1.6 | Biomechanics | 28 |
| 2.1.7 | Other research | 29 |
| 2.2 | Overview of the motion transformation methods | 31 |
| 2.2.1 | Motion transformation without dynamics | 31 |
| 2.2.1.1 | Motion warping | 32 |
| 2.2.1.2 | Motion signal processing | 32 |
| 2.2.1.3 | Fourier principles in animation | 32 |
| 2.2.1.4 | Interactive behavior adjustment | 33 |
| 2.2.2 | Motion transformation with dynamics | 33 |
| 2.2.2.1 | Early work of Witkin et al. | 33 |
| 2.2.2.1.1 | Spacetime constraints: | 33 |
| 2.2.2.1.2 | Spacetime windows: | 34 |
| 2.2.2.1.3 | Motion blending with spacetime: | 34 |
| 2.2.2.1.4 | Hierarchical spacetime control: | 34 |
| 2.2.2.2 | Other interesting works | 35 |
| 2.2.2.2.1 | Global search algorithm in spacetime optimization context: | 35 |

| | | |
|-----------|--|-----------|
| 2.2.2.2 | Interpolating between key frames with spacetime optimizer: | 35 |
| 2.2.2.3 | Recent work by Popović | 36 |
| 2.3 | Scope of this thesis | 36 |
| 3 | Physically based motion transformation system | 38 |
| 3.1 | Principles of the physically based motion transformation | 38 |
| 3.1.1 | Problem setup | 38 |
| 3.2 | Implementation of the physically based motion transformation system | 39 |
| 3.2.1 | User interface | 40 |
| 3.2.1.1 | Animation viewer | 41 |
| 3.2.1.2 | Animation playback control bar and other main window controls | 42 |
| 3.2.1.3 | Problem details window | 44 |
| 3.2.1.4 | DOF details window | 47 |
| 3.2.1.5 | 2D graph window | 49 |
| 3.2.2 | Input/output | 51 |
| 3.2.2.1 | Original MTS motion data file format | 51 |
| 3.2.2.2 | Input data | 55 |
| 3.2.2.3 | Output data | 56 |
| 3.2.2.4 | Types of DOF basis functions used | 57 |
| 3.2.2.5 | Loading animations produced by Electronic Arts motion capture studio | 57 |
| 3.2.3 | Motion mapping and transformation | 58 |
| 3.2.3.1 | C++ classes representing objective function components and constraints | 59 |
| 3.2.3.2 | Mapping from the full character onto the simplified one | 62 |
| 3.2.3.2.1 | What is exposed to the user | 62 |
| 3.2.3.2.2 | Problem setup details and implementation | 63 |
| 3.2.3.3 | Interface with SNOPT | 64 |
| 3.2.3.4 | Spacetime motion transformation proper | 66 |
| 3.2.3.4.1 | What is exposed to the user | 67 |
| 3.2.3.4.2 | Problem setup details and implementation | 68 |
| 3.2.3.5 | Mapping from the simplified character onto the full one | 68 |
| 3.2.3.5.1 | What is exposed to the user | 69 |
| 3.2.3.5.2 | Problem setup details and implementation | 70 |
| 4 | Video games and motion transformation | 72 |
| 4.1 | Using physically based motion transformation in video games | 74 |
| 4.1.1 | Dynamic motion retargeting | 74 |
| 4.1.2 | Real time motion mapping | 76 |
| 4.1.2.1 | Motion “frequency” | 76 |

| | | |
|----------|--|------------|
| 4.1.2.2 | Direct DOF substitution | 78 |
| 4.2 | Experimenting with models | 80 |
| 4.2.1 | Hopper | 80 |
| 4.2.2 | Biped | 82 |
| 4.2.3 | Walker | 85 |
| 4.2.4 | Torso | 89 |
| 4.2.5 | Guidelines for designing models suitable for DDS | 89 |
| 5 | Conclusion | 99 |
| 5.1 | Review of objectives | 99 |
| 5.1.1 | Near real time motion retargeting | 100 |
| 5.1.2 | Real time motion upmapping and motion compression | 100 |
| 5.1.3 | Applying motion transformation to submodels | 101 |
| 5.1.4 | Motion transformation engine as the last stage of the mocap pipe . | 101 |
| 5.2 | Review of results | 102 |
| 5.2.1 | Near real time motion retargeting | 102 |
| 5.2.2 | Real time motion upmapping and motion compression | 103 |
| 5.2.2.1 | Real time motion upmapping | 103 |
| 5.2.2.2 | Motion compression | 103 |
| 5.2.3 | Dynamic motion construction with submodels | 104 |
| 5.3 | Future research directions | 104 |
| 5.3.1 | Applying DDS concept to the downmapping stage | 104 |
| 5.3.2 | Replacing SNOPT with the real time or near real time solver . . . | 105 |
| 5.3.3 | Motion transformation system as the last stage of mocap pipe . . | 105 |
| 5.4 | What lies ahead | 108 |
| A | Disney principles of animation | 111 |
| B | Mathematical programming | 114 |
| B.1 | Linear programming | 115 |
| B.2 | Nonlinear programming | 115 |
| C | List of C++ modules in the MTS | 118 |
| D | List of animation clips online | 125 |
| | Bibliography | 127 |

List of Figures

| | | |
|------|--|----|
| 1.1 | Human muscle system | 3 |
| 1.2 | Hierarchy of the human body | 4 |
| 1.3 | Throwing mass around | 5 |
| 1.4 | Art of keyframed animation | 7 |
| 1.5 | Non-realistic keyframed animation | 8 |
| 1.6 | Exaggerated motion | 10 |
| 1.7 | Even more exaggerated motion | 10 |
| 1.8 | Animation channels | 12 |
| 2.1 | Disney animator at work | 16 |
| 2.2 | Pluto's sketches | 17 |
| 2.3 | Disney workaholic | 19 |
| 2.4 | Bounder | 27 |
| 2.5 | Sensors | 27 |
| 2.6 | Actuators | 28 |
| 3.1 | Global MTS module diagram | 40 |
| 3.2 | Main window | 41 |
| 3.3 | Problem details window | 46 |
| 3.4 | DOF details window | 48 |
| 3.5 | 2D graph window | 49 |
| 3.6 | DOF class diagram | 57 |
| 3.7 | Objective components class diagram | 60 |
| 3.8 | General functions class diagram | 61 |
| 4.1 | Motion components | 77 |
| 4.2 | DDS upmapping | 79 |
| 4.3 | Hopper and Human | 81 |
| 4.4 | EAHuman wide run | 83 |
| 4.5 | DDS-upmapped EAHuman wide run | 84 |
| 4.6 | Walker and Human | 86 |
| 4.7 | EAHuman criss-cross run | 87 |
| 4.8 | DDS-upmapped EAHuman criss-cross run | 88 |
| 4.9 | Torso and Human | 90 |
| 4.10 | The Torso model | 91 |
| 4.11 | Normal EAHuman run | 92 |

| | | |
|------|--|-----|
| 4.12 | Torso motion | 93 |
| 4.13 | Downmapped Walker motion | 94 |
| 4.14 | Transformed Torso motion | 95 |
| 4.15 | DDS-upmapped EAHuman combined motion | 96 |
| 4.16 | The DDS skeleton matching | 97 |
| 5.1 | Mocap filter's user unterface | 106 |
| 5.2 | Popović's system in a nutshell | 107 |
| 5.3 | Mocap motion filter | 108 |
| 5.4 | Realistic human faces | 109 |

List of Tables

| | | |
|-----|--|-----|
| 2.1 | Cartoon production statistics | 18 |
| 3.1 | Animation viewer control keys | 43 |
| 3.2 | Animation playback properties pane | 45 |
| 3.3 | UI controls of the 2D graph window | 50 |
| 3.4 | Actions available in the 2D graph widget | 51 |
| 3.5 | Preamble variables | 55 |
| A.1 | Animation principles | 113 |
| C.1 | MTS C++ modules | 124 |
| D.1 | Animation clips | 126 |

Chapter 1

Introduction

1.1 Motion as a research subject

The ability to move independently is the distinguishing feature of the objects in the animated world. Inanimate world “inhabitants”, like stones or mountains, do not move by themselves. In contrast, many inhabitants of the animated world are in constant motion. Simple organisms are in constant motion, searching for food. The motion of more complex organisms, like animals and humans, is based on many more factors.

In this thesis, we are not concerned with the motion specific to the *internal* structures of the living organisms. We will rather focus on the motion of the organism as a whole. To be exact, we will look into the motion of humans, or any other human-like organisms. We will be more interested in dynamic motion involving significant, sometimes extreme, forces and torques. Running and jumping are good examples.

Portraying human motion is one of the main things the movie industry deals with. Good actors are able to convey virtually any emotion with slight changes in motion pattern. Combined with strong facial play skills, the actor’s motion is a sufficient medium for expressing movie plot and ideas. Early works of Charlie Chaplin [38] can serve as an excellent example of what can be achieved in a movie by using the motion alone, without sound.

With proliferation of the personal computing and home entertainment systems (game consoles) portraying motion has become a key topic in the interactive entertainment industry as well. The dynamic nature of interactive entertainment imposes a different approach on motion directing techniques. Where movies offer us “canned motion”, static and unalterable by definition, computer games synthesize motion on the fly. The best example of synthesized motion is any game simulating sport activity. Such games as NHL 2001 [7] and FIFA 2001 [6] have a large collection of human motion samples inside, and use them to construct motion on the fly, depending on the user input.

Since games still have significant amounts of “canned motion” in them, there must be a way of obtaining that data and storing it in the game. For example, the most convenient way to obtain animation data for a soccer player, is to invite a real soccer star, ask him to do all necessary movements and capture them all. This process is called *motion capture* or simply *mocap*. Whether an optical, electromagnetic or electromechanical method of

capturing is chosen, we always end up with the array of static data, suitable for animating a game character, but which is virtually unalterable, see [62] for details.

The motion is synthesized in the game using a technique called *motion blending*. The static motion samples are blended together at run time. The animation sequence of a soccer player running is blended with jumping, and then blended with a hitting ball motion. The result looks like the player is running and then scoring a goal with a jump. Although the user controls the game character's behavior, the motion samples the player is animated with remain static. This produces two problems. The first is that motion constructed with blending static samples looks repetitive. The human eye can easily distinguish between such a motion and a "real" one. The other problem is the high price of motion capture. With sophisticated optical mocap hardware and requirements for motion data postprocessing¹ game designers are restricted to a preset pool of motion samples. Had the game designer decided to acquire additional samples or correct existing ones, he inevitably would have had contacted the mocap studio and asked for additional data, thus adding extra cost to the development process.

As we can see, editing the captured motion, or *retargeting* it (i.e. adapting characters motion to the changed conditions in the game) is a feature much wanted by game designers. It is also a very hard problem to solve and so far there is no successful commercial motion editing/transformation software available.² Our research focuses on two issues connected with the motion retargeting. First, we look into the possibility of implementing a motion retargeting engine in a game involving a significant amount of human animation. Second, we describe a method of storing motion in the compressed form and algorithm for its real time decompression. As a side issue, we also describe a method of dynamic motion construction with *submodels*. The submodel can be a part of a human skeleton, for example. If we have a submodel for the lower torso and a submodel for the upper torso, we can combine animations for both of them in real time, constructing new motion on the fly.

Although initially our research was about the possibility of making motion retargeting real time or near real time, we have not obtained significant results in this area. Instead, we devised techniques which will help game designers to increase the variety of the game characters' motion. These are the above-mentioned motion compression and dynamic motion construction with submodels.

1.2 Motion in nature

In order for living organisms to move in space, they must be able to generate forces necessary to perform motion. Animals and humans generate those forces with *muscles*, organs that transform energy released during chemical reactions into the mechanical energy. The human body has 640 skeletal muscles [42], some of them shown on figure 1.1. The human muscular system allows for a very wide range of movements, from pole-vaulting to playing the piano. When playing the piano and when pole-vaulting, the performer is focusing

¹In optical mocap images acquired by cameras require complex software and human intervention to produce joint angle values ready to be included in the game. See [62] for details.

²We mean only the captured motion here.

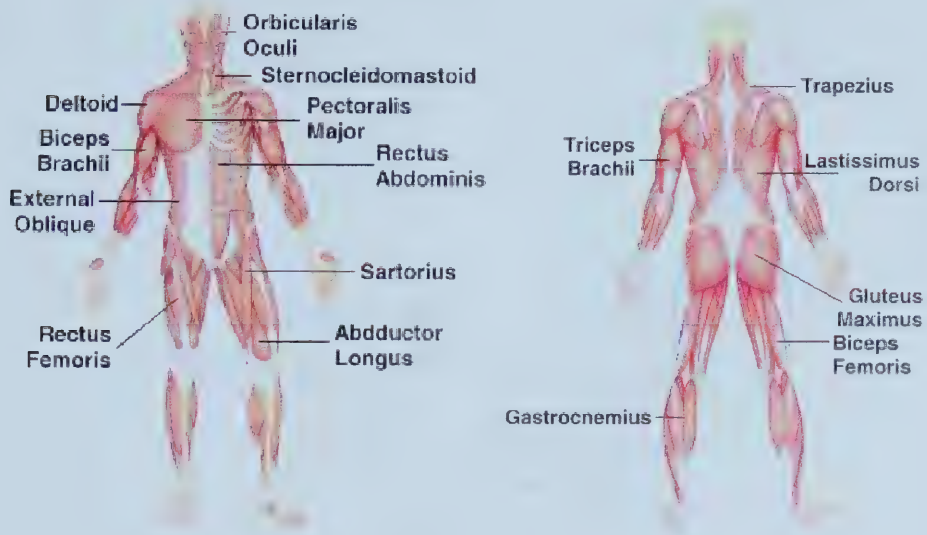


Figure 1.1: Some of the 640 muscles in the human body

on different aspects of motion. Musical instruments require precision and well-developed finger musculature, while pole-vaults require powerful lower body muscles, able to propel the whole body several meters high.

Any kind of motion, whether it is precise or high-powered, conforms to the “throwing the mass around” principle [18]. This principle states that motion in the animate world is characterized primarily by the *relative mass displacement*. Using this principle we can classify various kinds of motion. Highly dynamic motion is characterized by the large mass displacement values, while non-dynamic motion has mass displacement close to zero. If we present the human body as a hierarchy of nodes (see figure 1.2), we can measure mass displacement of the particular k -th node using this formula from [78]

$$E_k = \int_i (\mathbf{p}_i - \bar{\mathbf{p}}_i)^2 \mu_i dx dy dz$$

where \mathbf{p}_i is i^{th} body point in the beginning of motion, $\bar{\mathbf{p}}_i$ is i^{th} body point at the end of motion, and μ_i is an i^{th} point’s mass. Figure 1.3 displays two kinds of motion, bending the torso and bending the index finger. This is a visual demonstration of the difference between movements causing small and large mass displacements. The relative mass displacement criterion used in the motion transformation system described in [78] played an auxiliary role there. Popović employed the mass displacement metric as a way to measure difference between two character poses. In contrast to this, our research uses that principle as a theoretical ground. Pushing off from that principle, we developed a method for real time motion *upmapping*. We will provide more examples proving the importance of the relative mass displacement principle in the following chapters.

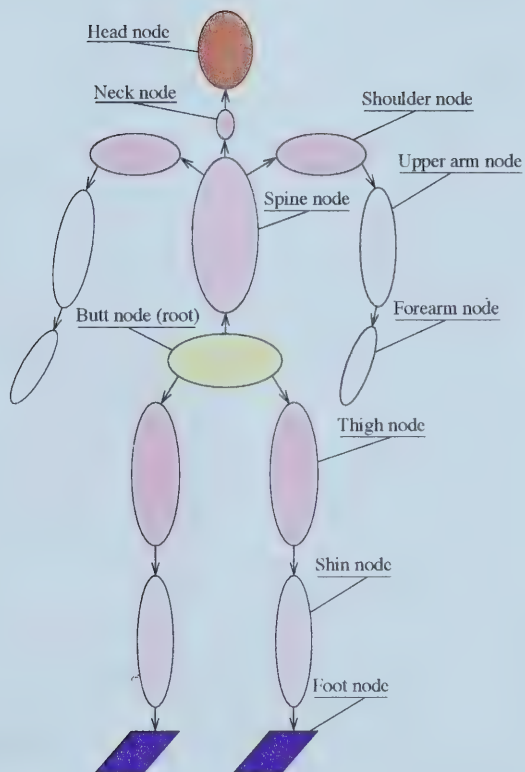


Figure 1.2: Human body as a balanced tree with butt as a root. This representation is common in computer graphics and animation.

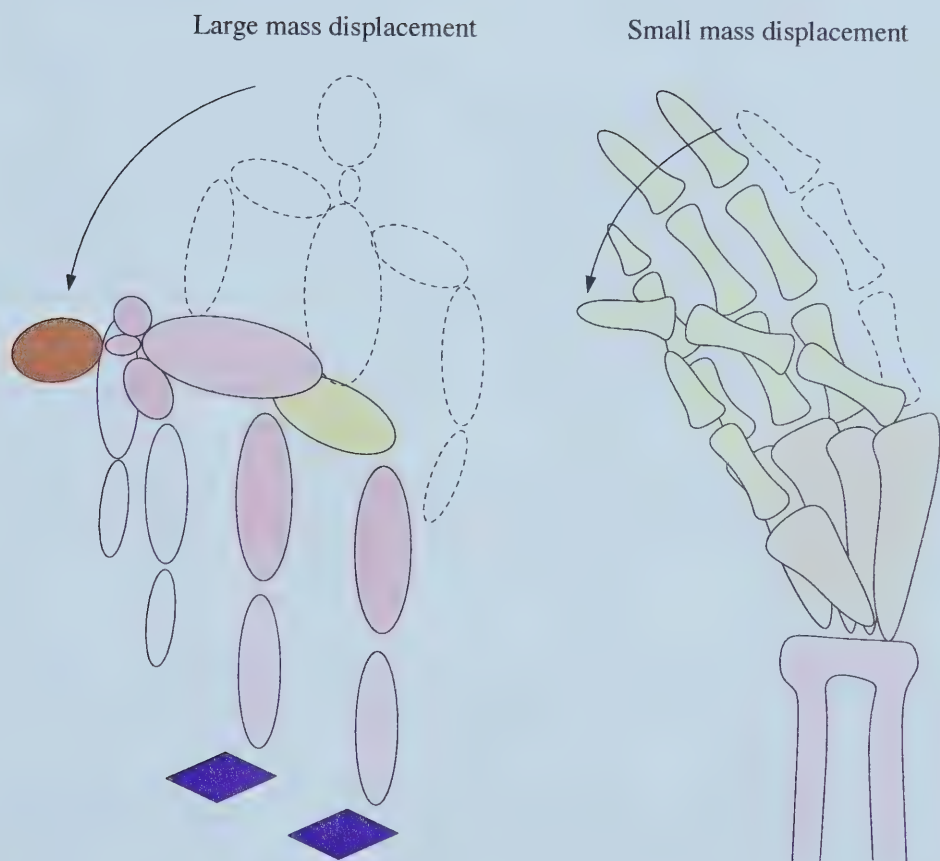


Figure 1.3: Bending torso vs. bending index finger

1.3 Motion in computer graphics

Computer graphics traditionally has had a very wide spectrum of research directions. In recent decades computer animation has become one of the hottest topics, especially 3D animation. As 3D modelers and animators try to reflect the laws of the real world in their work, they use the same underlying principles as nature itself. For example, dynamics simulation systems use the approximate representation of the muscular and skeletal structure of the character. In such systems, the core idea is numerical simulation of the natural laws of motion. The closer the model is to the real character, the higher the quality of the motion that can be produced.

3D animation techniques have roots in the traditional animation world. Both employ such notions as exaggeration and timing [84]. Both use *keyframing* [95] as the primary method of specifying character motion. Keyframing, with its low-level motion control, shines when coupled with a highly experienced animator, but even then it could have noticeable defects. These defects tend to appear when character exerts significant forces during sufficiently complex motion. Jumps followed by swinging on hands, or runs coupled with torso rotation about vertical axis are examples of the motion fairly hard to create with keyframing. The 3D animated movie “The Killer Bean 2: The Party” [56] demonstrates both strong and weak points of keyframing. Figures 1.4 and 1.5 are the stills from that movie. Figure 1.4 shows a strong side of keyframing. You can see what an experienced animator can achieve, making the character very expressive without applying significant effort. Slight exaggeration makes the character look theatrical, adding some charm and grace. On the other hand, the screenshot shown on figure 1.5 demonstrates less pleasant artifacts in the Bean’s motion. His jumping and swinging on a hook look anything but realistic. Precise, smooth and *realistic* animation of a body hanging on a hook and swinging with fading oscillations is a daunting task for a keyframe animation system.

Contrary to low-level motion control with key frames, dynamics simulation-based systems always produce very smooth and realistic motion. The underlying numerical engine ensures that Newton’s laws of motion are observed for every frame of the animation. Constraints-based motion transformation systems described in [30], [58], [103] and [79]. These dynamics simulators do not involve a *forward dynamics* [8] approach, they focus on changing the existing motion instead. One of these systems, namely the one presented in [78], is the subject of our research. Dynamics simulators can be thought of as systems of the opposite ideology in comparison with the keyframing animation packages. Dynamic simulation restricts animators, taking most of the responsibility of animating the character and replacing it with the convenience of the high-level motion control. Each approach has its advantages and drawbacks. The optimal solution tends to lie somewhere in between.

1.4 Motion in computer games

Good computer games always combine both the real and the unreal sides of life, with the unreal side being driven by the designer’s imagination. As an example of the realms of fancy we can mention the game “American McGee’s Alice” [4], where the visually



Figure 1.4: Killer Bean is warming up

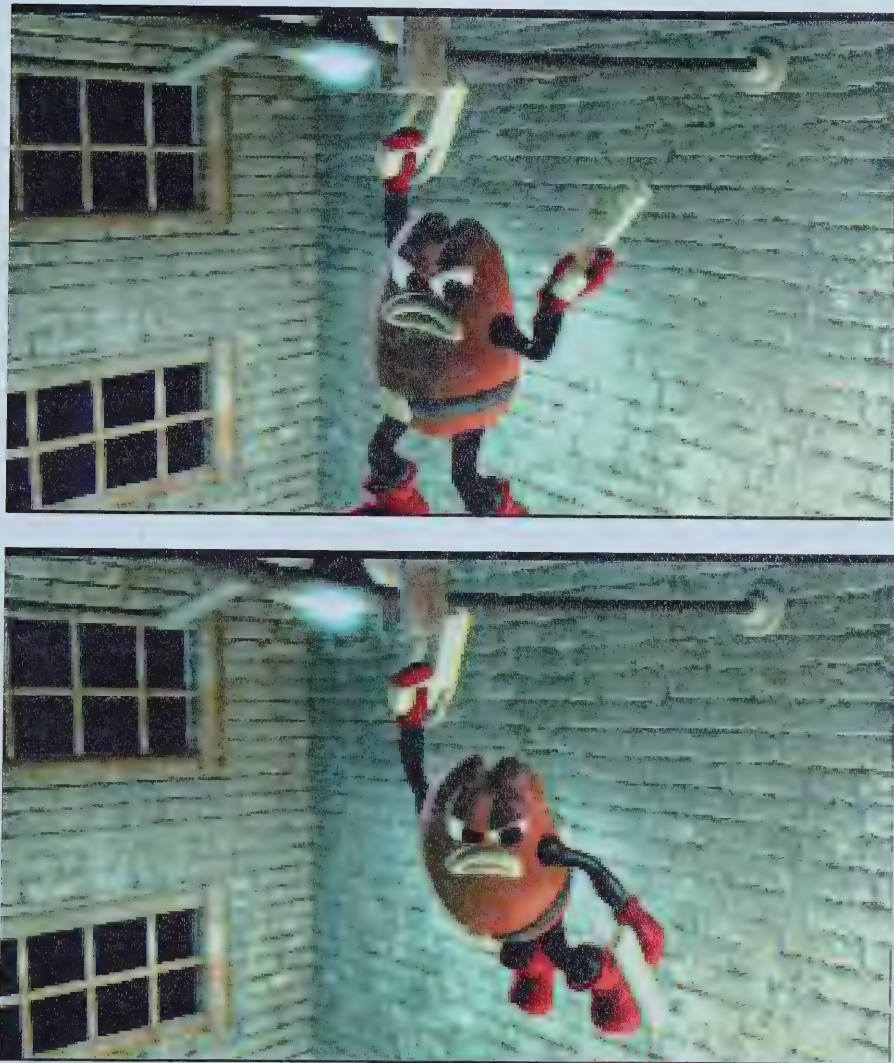


Figure 1.5: Killer Bean is hanging on a hook

stunning side of the game owes much to the outstanding art and the professionally made music score. In the other game of the same genre, “Clive Barker’s Undying” [5], both art and very expressive ambient sound effects are responsible for the “unreal”, “being in the dream” feel of the game.

The magic of the art and music is not enough alone to immerse player in the game. There must be elements of the real life in the game, to which player can “attach”, thus following the game’s plot. Common sense notions like gravity and three-dimensional space are present in most games.³ The best representation of the real side of any action/sports game is the motion. The way game characters move and interact with a player affects his/her perception of the game world. This is important for any first person shooter and is doubly important for sports simulator games. Hockey, baseball, and soccer games demand a lot of motion data, because these games are so dynamic and game characters perform all sorts of jumps and runs. Hence the better the motion quality is in such a game, the more absorbing and attractive it becomes. High-quality motion data in the sports game is certainly not the only factor determining success or the failure of the game, but it is one of the more important ones.

After observing and trying numerous types of computer games we can introduce the notion of the *motion LOD*, or *motion level of detail*. The metric for measuring motion LOD is the relative mass displacement. To explain the motion LOD concept we will refer again to the pole-vaulting and playing the piano examples from section 1.2. Playing the piano involves very precise finger movements, while all the other parts of the player’s body remain relatively still. This is an example of motion with a high level of detail. Consequently, if we measure relative mass displacement over the whole player’s body during this kind of motion, we will get a very low value. Unlike piano playing, vaulting with a pole involves a relatively large mass displacement value. Hence, we can say this kind of motion has a low level of detail. We do not care if the sportsman’s mouth is open or closed during the vault, and it is not at all important if the sportsman’s thumb on his right hand touches the index finger or not.

Motion with low LOD tends to prevail in computer games. There are two reasons for that. One is that high LOD motion will cost additional resources to implement, since the more complex character model needs to be built and animated. The other is that nobody cares about the finger positions of the soccer player when he is about to score a goal, it is impossible to notice this anyway, given that the player’s point of view tends to be quite far away from the character’s fingers. The same holds true for the first person shooters (FPS), where characters are either shot from a distance, or fight with the player without waving their fingers or moving their eyebrows. Such small details are insignificant in a fight. In other words, since the exaggerated movements are so valuable in the games, the low LOD is beneficial there. The lack of fine details in the low LOD motion is very well masked by exaggeration.

Therefore, you will not see really detailed characters in current 3D games, whether they are FPS or sports simulators. Consider such a complex object as a human hand. 3D modelers usually simplify a character’s hand and fingers, resulting in a hand which does not resemble a *real* hand. Animators, in their turn, compensate for such a simplification

³The game “Descent” [88] is the most notable exception, regarding gravity.

with a motion exaggeration. The games “Max Payne” [34] and “Undying” [5] are good examples of such a technique. Here we present two screenshots from the game “Undying”.



Figure 1.6: Aaron is searching for a book, brandishing his arms

Figure 1.6 depicts one of the game characters, Aaron, looking for a book in his house library. In real life you probably will never see somebody looking for a book on a bookshelf, swinging his/her hands frantically, but Aaron does. This is a simple trick compensating for the simplification of his hands. Indeed, how can the animator show that Aaron is looking for a book, if his (Aaron’s) hands are simplified objects? Well, the animator makes the motion exaggerated, so now it is obvious to everyone that Aaron is looking for a book.



Figure 1.7: Howler’s lethal jump. Guess where he would like to land.

The world of “Undying” is inhabited not only by humans. Human-like creatures called

“howlers” demonstrate another example of motion with a low level of detail. A howler chases its prey by leaping or jumping at it. If you look at figure 1.7, you will understand why the howler’s model is relatively simple. With the kind of activity a howler usually performs it is not necessary to make his model very detailed.

As we can see from the examples above, model simplification plus motion exaggeration serves well for the purpose of making 3D computer games simpler to design and maintain.

1.5 Motion transformation and computer games

Besides the game code, which usually consists of AI and graphic engines plus some user interface, a 3D game has large amounts of multimedia data inside. The most widespread medium for storing games, the compact disc (CD-ROM) can store only about 650 megabytes, which is no longer considered a large amount of data. For PC games this is not a big issue, as consumers always have large hard disks on their computers, and the game can split the necessary amount of data across several CD-ROMs. However, for game consoles, like Sony Playstation [33] and Playstation 2 [32], this is impossible to do as consoles currently do not have hard disks.⁴ Hence every possibility to compress data must be explored and used in the game. Sound and music can be compressed with MP3 codec [36]. “Alice” [4] is an example of such an approach. Images like textures and bitmaps could be compressed with the JPEG image compression technique [43]. What is left uncompressed? The motion data of course. At the moment the character animation sequences are stored uncompressed as a set of *animation channels*, shown on figure 1.8. Because the game character’s body is significantly simplified in comparison with a human body, the number of channels is rarely exceeds 80. Even with the number of animation channels below 80, the animation data can take a huge amount of space if the game involves hundreds of animations. This is exactly the case with some sports games, like soccer or baseball.

Our research focused on motion editing/retargeting methods, but the results we obtained allow us to propose a solution for the motion data compression issue. The foundation for our research in motion compression and retargeting is the physically based motion transformation system, described briefly in [79] and in detail in [78]. Although this system is unfit for production use yet, it is capable of editing captured motion, and it is not restricted to captured motion only.

From the point of view of the possible motion retargeting in the game, the system is too slow at the moment. Our research analyzes the system functionality and offers ways to speed things up. Given that the system has gained real time performance, both the motion storage issue and the need for recapturing performer’s movements for every change in the game character’s motion are solved. Unfortunately, the real time or near real time motion retargeting with Popović’s system requires a major redesign of the core numerical solver module, which is far beyond the scope of this thesis. The modifications of the system performed in our research only touch the surface of the problem. Much more work is required to make this system usable for the real time motion retargeting.

⁴This is fixed in Xbox [31], an upcoming console from Microsoft, and will be addressed in Sony Playstation 2 [32] as well.

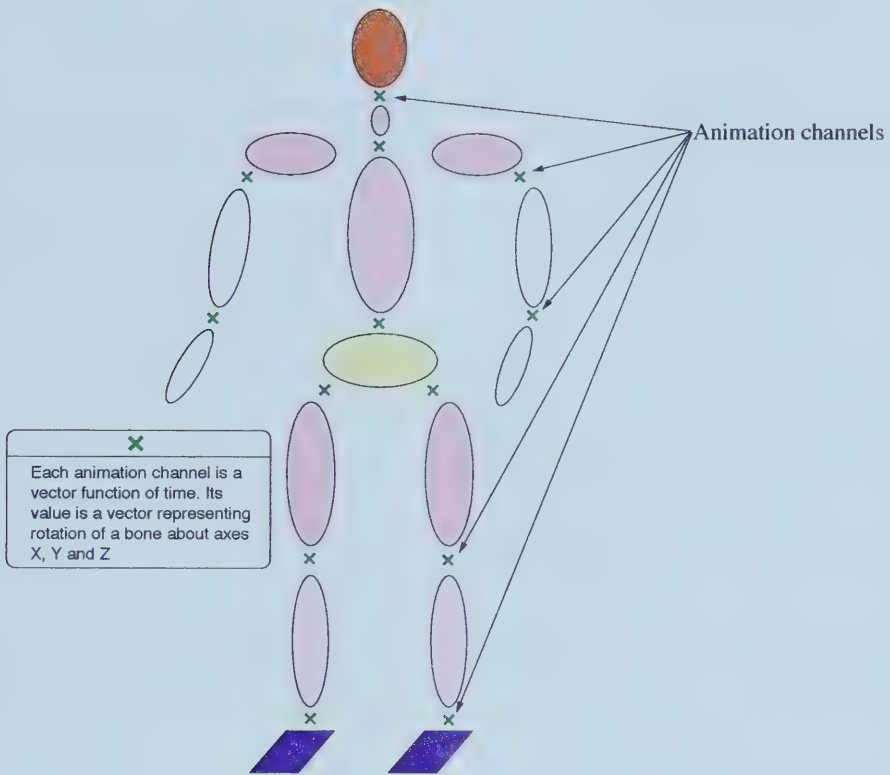


Figure 1.8: Motion is stored in a game as a set of animation channels

However, we can exploit Popović's system in another way. One of the stages of the motion transformation pipeline in this system is motion *downmapping*, i.e. transferring motion from the full character model onto the simplified one. While the full character might have 80 animation channels, the simplified one could have as few as 16 channels. This gives us a theoretical motion compression ratio of 5 to 1. The only obstacle to inclusion of compressed motion in a sports game is a lack of the real time motion *upmapping* algorithm. This algorithm should transfer motion from the simplified model to the full one. Our research deals with this problem and proposes such an algorithm. Exploiting the motion LOD concept introduced in section 1.4 and some other features of the motion presentation in the motion transformation system and computer games, we were able to restore motion in real time, mapping it from the simplified model to the full one.

In addition to the above two issues, our research also considers dynamic motion construction. Using the same idea as in the real time motion upmapping algorithm, we show that it is possible to combine the motion of the several submodels into the full model motion. For example, we could have a set of animations for the upper torso and another set for the lower torso. Combining motions from both sets, we can produce new animations on the fly. This concept can also help in saving space taken by the motion data.

1.6 Thesis outline

Chapter two deals with the existing methods of motion transformation. Such concepts as motion warping and spacetime constraints are considered. It gives a fairly broad overview of what was happening in the area of motion transformation during the last fifteen years.

Chapter three is a description of the physically based motion transformation system by Popović. A detailed explanation of the motion transformation algorithm is given, followed by an analysis of the system modules. Some source code listings and screenshots illustrate the system functionality.

Chapter four is about the research we performed on the motion transformation system. It describes real time motion upmapping concepts, explains direct DOF substitution (DDS) idea, provides examples of the transformed motion and also elaborates on the limitations of our approach. As our method of motion upmapping is based on the direct substitution of the animation channels, the models (skeletons) involved in the process must have certain specific properties for the algorithm to succeed. In addition to the main contribution, real time motion upmapping, this chapter also sheds some light on the possibility of the real time motion retargeting and describes dynamic motion construction.

Chapter five concludes this thesis. It reviews the contribution we made in the area of motion transformation research, as well as the benefits and limitations of our approach. Some thoughts about the future of motion transformation in computer games are there as well. Finally, the possibility and prerequisites of making the sports game replicate the TV translation of the real life match are discussed.

Chapter 2

Background

As the epigraph for this thesis suggests, motion is the way to convey the plot and the characters' emotions in cinematography. In computer graphics, motion often serves the same purpose. Although, given the synthetic nature of computer graphics, it may be used for other purposes; for example, in TV commercials or in business presentations. Therefore, we could separate motion in computer graphics into natural and synthetic, where natural is concerned with the expression of motion in nature, and synthetic is everything else - flying 3D geometric primitives in a TV commercial or sliding financial charts in a presentation.

Of course, the real challenge is the kind of motion associated with complex objects existing in the real world, particularly, the animation of humans and animals. In this kind of animation it is extremely hard to express subtle changes in characters' emotions or mood. The problem stems from the fact that the human perception system is very sensitive to small details in motion, especially regarding the face and eyes. What is natural and easy to perform for a professional actor or actress, often becomes almost an insolvable task for a team of highly skilled animators. The art of animation turned out to be a full-scale artistic domain, with its own established authority (Walt Disney), a variety of styles, rivalry between competing schools and studios and a typical feature of any art - a good animator is the one who has both innate talent and great diligence in studying techniques and style of animation. As in any other form of art, only both of these components may lead to excellence in the art of animation.

The recent fast-paced invasion of personal computers, accompanied by the blossom of computer graphics as a science, led to substantial simplification of some stages in animation, for example, in inbetweening [90]. What has not changed, though, was the set of requirements for the animator. Whether creating a scene on a sheet of paper or on a computer screen, the animator is still supposed to control *everything* in motion, every minute detail. In other words, computers did not lower the bar for those wishing to master the art of animation.

Fortunately, things gradually started to change in that area. Recent research in motion synthesis and transformation could allow creation of simpler, "consumer-oriented" animation packages, offering high-level motion control and built-in *motion libraries*. The two following sections offer a concise overview of motion synthesis and transformation techniques. We will sketch the history of 2D animation, focusing on the Walt Disney

classic works. 2D animation is followed by the introduction of several techniques for computer-aided motion synthesis, from keyframing to the dynamic simulation methods. However, the main focus of this chapter is the motion transformation methods. Here we will give a more detailed analysis of various methods of motion transformation, including simpler methods without dynamics, as well as the more recent spacetime constraints approach which is actually the topic of this thesis.

2.1 Overview of the motion synthesis methods

Synthesis of motion is another word for animation, or a process of bringing the drawn character to life. The most often used computer animation technique these days is keyframing, which has its roots in cartoon animation. The basic principles of animation, discovered by Walt Disney¹ and his animation team, are still widely used, almost a hundred years after their discovery. They apply both to cartoon animation and to computer animation, whether it is 2D or 3D. However, compared with paper and pencil, the tools of trade of a conventional cartoon animator, computers offer more freedom and a much wider range of possibilities. They are good at simple keyframe automation tasks like inbetweening and coloring, but this is far from the exhaustive list of things they can do. Computational power can be exploited by the animator with various alternative animation techniques, from procedural animation down to complex physics simulation of the forward dynamic methods. These methods have been used as an addition to keyframing so far, but with the appearance of realistic, physically based motion synthesis/transformation methods, occasionally those alternative methods could be used as the primary motion synthesis technique.

We start our overview with keyframing, the most popular *kinematic* motion synthesis method. Then, we will cover another kinematic method - procedural animation. After that, we will give some background on the synthesis with dynamics, and the last technique we will mention is motion capture. We will also include two additional sections on biomechanics and robotics, showing their relation to the problem of motion synthesis.

2.1.1 Keyframe animation

Keyframe animation is the first discovered and the most widely used way of bringing drawn characters to life. The name of this method comes from the notion of a *key frame*, which is basically a frame from the animation sequence showing a character at some moment in time. All the animation sequence consists of a series of key frames. The task of the animator is to express the action of the character in this key frame series. The quality of the resulting animation directly depends on the animator's talent and experience. Complex animation scenes involving interaction of several characters or subtle emotion/mood changes in character may require years of training and studying of human/animal motion.

¹In fact, Disney was not the first one to discover those principles, there was a previous work of Winsor McCay, but Disney was the one who extended McCay's techniques and built a successful business selling animated cartoons. See [90] for details

After the series of key frames has been produced and approved, it is handed to the team of *inbetweeners*, whose task is to fill the gaps between the key frames. Computers can easily handle this task too, using splines to interpolate frames smoothly. The other production work like coloring is perfect for computers as well, but the core part of character animation is still the hand drawing of the key frames. No inbetweening and post-production processing can improve the situation if the original set of key frames was poorly designed.



Figure 2.1: Disney animator Norm Ferguson searches for the right expression

Figure 2.1 shows the delicate process of a key frame creation. Sometimes the animator has to use a mirror to convey drawn character's emotion convincingly. Before drawing key frames themselves, animators *sketch* characters, trying to capture the best expressions or poses, conveying character's personality. A series of sketches presented in figure 2.2 shows one of the most famous “actors” in Disney cartoons - a dog named Pluto. Notice that the range of expressions for Pluto is wider than that of a real life dog, so the animator is faced with the task of creating expressions unmatched in nature (laughing dog, crying mouse, etc.)

Looking at the above-mentioned importance of the key frames design, one can conclude that a keyframe animator has an absolute control over the character's motion. Every little

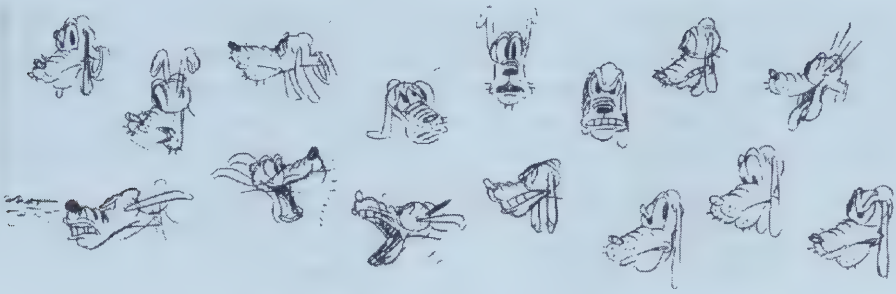


Figure 2.2: Some of the sketches of Pluto's face, demonstrating wide range of expressions

part of the character's body is controlled by the animator's pencil. Is it any wonder then that all the characters in the Disney cartoons (or animated cartoons produced by any other "traditional" animation studio) have a rather simplistic look. You will not see tiny wrinkles on Mickey Mouse's face or jags on Snow White's nails. The reason is not the absence of wrinkles on a mouse's face or jags on the young girl's nails (although this could seem to be a plausible explanation) but in the nature of the animator's work. A series of key frames represent *motion* in first place. Every additional detail in a keyframe means additional work for the animator. This is why deer in *Bambi* are not beautifully rendered 3D animals with their spotty coats and velvet ears, but are simply colored flat figures.

Recent involvement of computers in production of animated cartoons led to improvement in the overall picture quality, noticeable in such latest Disney animation works as *Aladdin* [74] and *The Lion King* [75]. With great coloring and background drawings, these Disney cartoons look smooth and impressive, although the animated 2D characters are basically the same as 70 years ago. After all, cartoon characters are not supposed to be photorealistic; motion is what it is all about.

Still, there exist experimental techniques which allow animation of the photorealistic images. For example, Litwinowicz and Williams proposed a method for animation of images with simple line drawings [57]. It seems that this method is of pure academic interest, though, since most questions related to the production of animated cartoons are left unanswered. The issues of integration of the Litwinowicz/Williams method into the production pipeline of an animation studio are yet to be explored. On the other hand, movies featuring talking animals [73] [72] would have been impossible without an animation technique similar to that.

Absolute motion control in keyframe animation is both a blessing and a curse. While not allowing the creation of photorealistic characters², it compensates for that with the absence of limits on the kind of motion that can be created. It is analogous to programming computers in assembler. One has total control over everything and one can create programs unmatched by speed and size, but the process itself is tedious and requires quite a bit of experience (and talent is certainly involved, too). Another issue in keyframe animation is that the quality of animation is proportional to the number of

²This statement has nothing to do with computer aided 3D animation, it is about traditional 2D animation, where every key frame is drawn manually.

key frames produced [25]. Therefore, there is a lower limit on the number of key frames, which should not be violated. Otherwise the motion will not look smooth and natural.

To illustrate the point and show the amount of work done on the full-scale animated cartoon we will cite an instance of animation statistics, taken from [90]. Table 2.1 gives an idea of how many drawings should be produced for an 80 minute animated cartoon.

| Drawing type | Number of drawings |
|------------------------|--------------------|
| Inspirational sketch | 1000 |
| Story sketch | 75000 |
| Layout drawings | 22000 |
| Animator drawings | 576000 |
| Inbetweener drawings | 1036800 |
| Assistant drawings | 345600 |
| Finished cels | 460800 |
| Subtotal | 2517200 |
| Miscellaneous sketches | 2000 |
| Total drawings | 2519200 |

Table 2.1: Production statistics table, demonstrating amount of work required for an 80 minute animated cartoon

Given these data it was not surprising to find overworked animators at their tables at the Walt Disney studio. One of those workaholics is depicted on figure 2.3.

Fortunately, today’s computer technology has something to offer to classic 2D cartoon animation. Fekete et al. proposed a system for automating most tedious tasks in 2D animation [35]. Combining pressure-sensitive drawing input pads with simple image processing, this system allows an animation studio to speedup the animation pipeline and to do most of the image processing without paper or *cels*³.

We conclude our overview of the keyframe animation by mentioning the list of all the basic animation principles discovered and formulated by the Disney studio animators. There are twelve of those, you can see them together with a short description of each in table A.1, appendix A.

2.1.2 Procedural animation

Procedural animation, using a set of functions governing motion, is probably the first to come to mind when thinking about possible applications of computers in that area. Although it is not of much help in adding necessary emotional nuance to the character’s motion, it will help in those cases where the laws of mechanics prevail over the animator’s creativity. For example, animating linked parts of machines does not involve anything beyond following the obvious rules for drawing interconnected details of machinery. Here, computers could help animators to automate this task. When the connection scheme of the details inside the machine is known, it is possible to calculate the position of the

³Cels are transparent celluloid sheets used in cartoon animation studios for certain drawing and coloring tasks where the transparent medium is required.

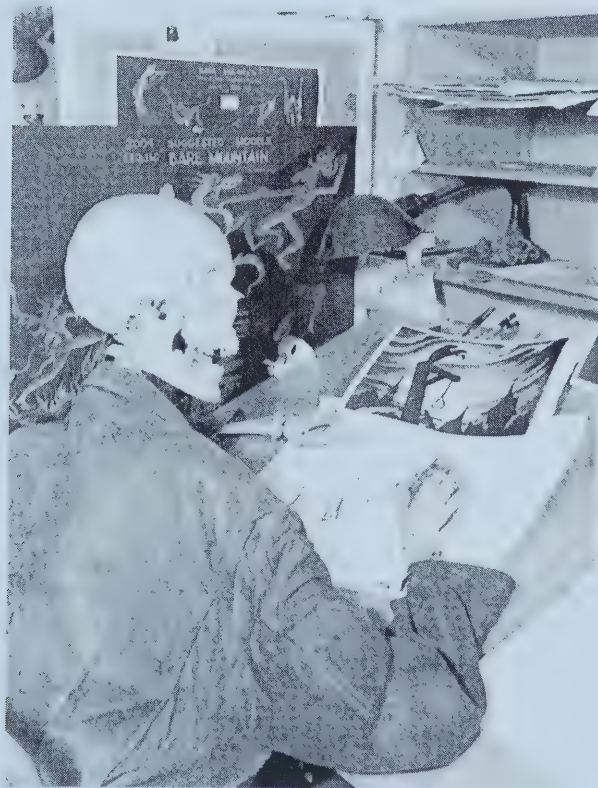


Figure 2.3: This animator has drawn too many key frames

various machine parts at any moment in time. Hence, animation turns into programming - the animator describes the connections between machine details and possibly their shape and the computer does all the rest.

Procedural animation is not restricted only to animating machinery. A particle system [81] is another example. The motion of each individual particle in the system can be described as a function of time, particle number, weight, color and any other property necessary. We can present a school of fish as a particle system, where each individual fish's global position and orientation is defined by the function set by the animator. In addition, individual animation for each fish (swimming movements) can be done by keyframing or by a physics-based simulation, which is even better.

Abstracting procedural animation even higher, to the level where animated subjects are given orders *what* to do, we shift into the area of *behavioral* animation. An example of this kind of animation system is described in a paper by Blumberg and Galyean [19]. In this system emphasis was made on the real time VR applications. The paper is sparse on details. One may suggest they employed procedural animation for low-level control of their virtual creatures. Also the paper by Tu and Terzopoulos [91], regarding animation of fish, has some procedural animation techniques, although it should be rather called a behavioral animation system, similar to the system made by Blumberg and Galyean.

Applying a knowledge of psychology and linguistics, Cassell et al. [27] built an interesting procedural animation package that animates characters in a conversation, depending on *what* they have said. Of course, the dialogs were machine-generated, to avoid the necessity of adding lexical and semantic parsers to the system. Analyzing speech data, the system was able to control characters' arm and face muscles, depending on certain words in the speech flow and punctuation signs. As all the other procedural animation systems, Cassell's system does not use any dynamics. All the motion is scripted and is purely kinematic.

To some extent, the animation system by Lee et al. [53] can be classified as a procedural animation system. Some notion of physics and biomechanics is presented, but it is all hidden in the scripting logic of the system. One can say that this system has "built-in" physics and requires only some parameters, like the character's geometry and strength, weight mass and lift height. After all those parameters are set, the animation of a human lifting a weight is automatically generated.

Overall, procedural animation tends to be an auxiliary tool in the animation studio. In terms of the computational power required, procedural animation in most cases is cheaper than any motion synthesis technique involving forward dynamics.

2.1.3 Dynamic simulation

After looking at the previous two sections, the reader is probably already asking why the straightforward approach of numerical motion simulation has not been mentioned yet. The answer is that the numerical simulation of complex mechanical systems requires significant computational resources. The complexity of the mathematical formulation of Newton's laws of motion prevented such systems to be implemented on consumer-level hardware. Today, the computing power of personal computers allows dynamic simulation modules to be included in the common 3D animation packages.

Numeric simulation of motion can be divided into two major areas - forward dynamics and optimal control. Since forward dynamic simulation, spacetime optimization and related methods have much in common and are extensively used in other animation techniques, we cannot draw a clear boundary here. Optimal control was used in the inbetweening phase of a keyframe animation method by Brotman and Netravali in [24]. Hodgins employed forward dynamics plus higher level control algorithms in her animation of a runner, a cyclist and a gymnast doing a handspring vault [48]. However, neither of them used pure forward dynamics or pure optimization.

One can also think of dynamic simulation as of a special case of procedural animation. Indeed, there is no conceptual difference in both animation methods. In procedural animation, the character's motion is kinematically scripted. In case of dynamic simulation it is scripted as well. The difference is inside the scripts. For procedural animation anything the user wants can be scripted but for dynamic simulation the scripts can contain only the logic of Newton's laws of motion.

2.1.3.1 Forward dynamics

Forward dynamics is the most obvious way of modeling motion mathematically. When deciding how to represent the motion of a complex character in 3D space this method is probably the first one that comes to mind. The core idea of forward dynamics is to calculate the acceleration of the character's body segments for each frame in the animation sequence, followed by numerical integration to determine motion trajectories. The word "forward" in the name for this method comes from the fact that for every frame acceleration, velocity and position data depend on all the previous frames. The process of computation goes "forward" from the first frame to the last. Forward dynamics is an excellent way of generating plausible and reasonably high quality animations for a system of interacting rigid or flexible bodies. A good introduction in forward dynamic simulation is given in [8]. There is also demonstration software by Melax [61], namely the part of it demonstrating behavior of rigid objects under the influence of external forces.

Early work in computer animation research dealt only with systems of *passive* objects; that is, objects driven only by external forces. The research focus was the simulation of passive systems of rigid or flexible bodies, where only the external forces exist. One of the earliest papers considering animation of multisegment rigid bodies with dynamic simulation was presented by Green and Armstrong [3]. This paper was accompanied by the similar work of Wilhelms and Barsky [98]. Among most prominent researchers in this area are David Baraff, Alan Barr, Andrew Witkin, Demetri Terzopoulos and Brian Mirtich.

Baraff is well known by his series of papers on collision detection and response in rigid and flexible body systems [11], [12], [13], [15], [14]. Barr made significant contributions in the development of various constraint-based dynamic simulation methods [102], [16], [76]. He also contributed to research in geometric collision detection between curved surfaces [46]. The multi-body collision detection algorithm presented by Barr et al. in [87] features detection of multiple simultaneous collisions and returns the sample of points from the bodies' contact region with user-defined density. Witkin's most well known work is his groundbreaking concept of spacetime constraints [103]. He also contributed to many

physics-based simulation research areas, ranging from fast rigid body simulation to the latest papers in constrained spacetime optimization methods [102], [79], [104], [105], [15], [77]. Terzopoulos contributed to inelastic, deformable body simulation [89], rigid body deformation simulation [63], applications of dynamic simulation in behavioral animation [91], realistic facial animation [54] and animation through automated locomotion learning [44], and other research areas. Mirtich developed a clever technique of asynchronous dynamic simulation of a large number of colliding rigid bodies [65]. By maintaining separate *virtual time* for every body being simulated, his algorithm sidesteps the tough problem of synchronous fallback typical for such simulation systems. As a result of that his algorithm is capable of dealing with a very large (up to several hundreds) number of interacting objects in real time.

Recent work by Chenney and Forsyth [29] allows for the creation of *many* animation sequences, all of them satisfying a set of constraints established by the user. The animation system of Chenney and Forsyth has a rigid body simulation engine, and can be used to generate wide range of plausible, realistic animation sequences. The user can set his/her preferences in the form of constraints, and then run the system. In its turn, the system will sample the space of motion sequences using the Markov chain Monte Carlo (MCMC) algorithm. After sampling, it will select those animation sequences that satisfy user's constraints. The user is given the freedom to select the best among them, according to his/her taste.

Decoupling geometry and dynamics can lead to significant increase in simulation speed, as was discovered by Pentland and Williams in [70]. Their physics-based non-rigid body simulator splits geometry from dynamics in the presentation of objects to be animated. This results in a simulation speed close to real time, which was impossible to achieve with a traditional approach, where geometry and dynamics formulations were identical.

There are many more works in the area of passive system simulation, but unfortunately the “forward” nature of this sort of numerical simulation leads to serious problems when trying to generate motion for self-propelling objects (exerting internal forces to propel themselves in the environment). It is very hard to solve the problem of finding the *control algorithm* (*controller*) to regulate the proper timing and direction of those internal forces. The human body has a very complex system of muscles, constituting the unified ensemble. During locomotion, the human body employs many muscles producing forces acting on the skeletal bones with varying magnitude and direction. Implementing the similar controller for the muscles in the virtual character turned out to be an almost unsolvable task. Many researchers experimented with custom-built controllers, but the end result is that there are no methods for the automatic generation of a controller system which would allow automatic synthesis of desirable motion.

Jessica Hodgins is one of the most prominent developers of dynamic simulators with hand-tuned controllers. She and her colleagues successfully implemented a forward dynamics based motion synthesis system in [48]. Equipped only by their intuition and extensive experience in biomechanics and computer graphics, they solved the controller problem manually, building control algorithms by hand. The results were impressive but as in any other kind of motion research work with hand-made control algorithms it showed only the possibility of such an approach and not its feasibility for use in production environment. Other works by Hodgins include animation of legged locomotion [80] and

mapping motion to characters with different anatomy [47].

Van de Panne built a prototype of real time interface for controlling dynamic simulation of various characters [52]. The first attempt by Baecker to build gesture interface dated back to 1969 [9]. Unlike Baecker's, van de Panne's features a physics-based animation engine, so it is not a puppet-like system, it is rather a "push'n'pull" system where the user interactively controls forces applied to the character being animated. Although avoiding the problem of building proper motion controllers, this system requires a certain skill level from users. As the complexity of the character being animated grows, more training on the user's side is expected and eventually the user is faced with the necessity of programming the system.

McKenna and Zeltzer implemented a custom hexapod locomotion simulation system in [59]. Their paper notices the principal difference between forward dynamics and optimal control methods (more on this in section 2.1.3.2). They implemented the forward dynamic simulation engine for their six-legged walking model, and the issue of controllers was solved traditionally, by coding everything manually. The hexapod as a whole is controlled by the gait controller while each leg has its own motor program controlling step and stance phases of motion. The gait controller here plays the role of central nervous system of the creature, even though the creature does not have any sensors.

With respect to control, there is always a tradeoff between the flexibility of kinematic control and the realism of dynamic simulation [86]. Keyframing is so popular because it gives ultimate control to the animator while dynamic simulation restricts creativity by imposing the limits of classical mechanics. Hahn tried to combine the kinematic and dynamic methods of motion control in [45]. The idea is trivial - in this mixed kinematic/dynamic environment both types of objects are allowed. Dynamic objects are animated by a rigid body dynamic simulation engine, while kinematic ones are scripted by the user. This is a good example of mixing procedural/kinematic approach with dynamics.

Another example of mixing dynamics with a keyframing-like approach is demonstrated by work of Bruderlin and Calvert [25]. Their legged locomotion animation system is goal-directed. There is a set of 31 parameters which is used when deriving pose constraints for the character being animated. Everything between constraints is handled by the physics engine, making synthesized locomotion look realistic.

2.1.3.2 Optimal control

Given some initial conditions, forward dynamics derives motion trajectory step by step. Optimal control methods are the opposite - they start with the existing trajectory and transform it. The goal of transformation is to minimize a user-defined function that depends on the trajectory. Spacetime optimization is a pioneering optimal control concept that allowed for an automatic motion synthesis with a very high level of control. Spacetime optimization was introduced by Witkin and Kass in [103].

Spacetime optimization needs an initial trajectory to transform. Still, if we set the initial motion trajectories of the animated character's segments to the constant value (for example, zero) we will essentially generate motion from scratch. Hence we can see two sides of this method. The secret is to find the optimal motion trajectory, satisfying a set

of nonlinear constraints and minimizing some multivariate scalar function. So, strictly speaking, this is motion transformation. On the other hand, after transformation it produces a new motion; therefore, we can call it motion synthesis as well.

As with forward dynamics, we cannot use spacetime optimization in its pure form in complex character animations. Spacetime works fine with models consisting of four to five nodes, but transforming or synthesizing motion for a character's complexity close to that of a human body is virtually impossible. The spacetime optimization method does not scale up to complex models. When the character with a large enough number of joints performs a lengthy motion, the corresponding problem size makes it intractable for a numerical solver. More precisely, the larger the size of the problem, the closer to the solution must be the initial motion trajectory. By the nature of multivariate optimization problems, locality of minima in general increases with problem size. Hence, with complex characters we need to setup an initial motion trajectory sufficiently close to the desired result. This requirement emphasizes the transformational nature of spacetime optimization. Therefore, for complex models it is strictly a transforming tool, not one which synthesizes.

Synthesis of motion for simple characters has been done by the original authors of the spacetime concept in [103], with variations regarding trajectory search in [68] and trajectory basis functions in [58]. Until recently, all researchers were concerned with spacetime optimization on simple characters. Only the latest work by Popović and Witkin [79] made significant advances in the unexplored area of spacetime motion synthesis/transformation for very complex characters. We will present a more thorough analysis of the spacetime trajectory optimization research in section 2.2.

Methods derived from various dynamics simulation concepts, be it of forward or spacetime variation, are the most promising ones in terms of high-level motion control they can offer. Simulation of physics of motion is naturally appealing, because it could fulfill the animator's dream - control the character by setting goals of *what* it should do, without detailed explanation of *how* it should do that. There is still long way to go, though, to fulfill that goal.

2.1.4 Motion capture

Motion capture, or *mocap*, is a technically sophisticated and expensive way of cheating motion synthesis. Instead of creating motion by animating 3D character model inside the computer, motion capture uses the data captured from a live performer, be it a human or an animal. By capturing motion instead of synthesizing from scratch (keyframing/dynamics) or transforming (spacetime optimization), we avoid the problem of controllability of the synthesized motion. In mocap everything is under the control of the performer. Absence of the high-level motion control issues does not make mocap a heaven for animators. It has its own share of rough spots; they are just different from those of the 3D computer animation. One of the larger ones is price, another one (in optical mocap) is the tedious process of marker identification and postprocessing of motion curves. The price can be beaten by mass production; at least, in theory. Manual postprocessing of mocap data can be eased by applying automatic marker identification methods, for example.

Mocap systems are divided into three major categories, each having its unique set of features and peculiarities.

Electromagnetic trackers: Electromagnetic space position and orientation sensors are a standard attribute of most VR setups because they can supply position and orientation data in real time. A motion capture system with electromagnetic sensors consists of one emitter of the electromagnetic field and a set of sensors. The orientational aspect of the information returned by the sensors is of no interest for the purposes of motion capture. Only the 3D position of the sensors is captured in real time. Attaching sensors to parts of the performer's body, we can get a real time picture of the performer in 3D space. We can save it and use it later or we can apply the data to the model and observe a puppet on the screen, mimicking every performer's movement.

Electromagnetic motion capture has the advantage of being real time but it cannot be used in cases where large motion capturing volume is needed. Another problem with electromagnetic capture is its extraordinary sensitivity to the large metal objects close to the emitter. It is hard to do precise position tracking in a room where the ceiling has steel supporting rods. The precision of sensors quickly decreases as the distance between sensor and emitter increases. 2-3 meters away from the emitter, positional data gathered from sensor becomes "shaky", i.e. if the model is animated by the data from electromagnetic sensors in real time, its extremities start shaking. The last disadvantage is the tail of wires which the performer must carry, as every sensor should be wired to its data entry port in a data gathering computer.

Electromechanical suits: The suit for electromechanical motion capture looks like an exoskeleton, which the performer wears. The suit itself is a system of rods, connected with joints equipped with angular sensors. The sensors transmit a real time stream of angular data to the host computer. In terms of being real time the electromechanical mocap system is the same as the electromagnetic one. Unlike the electromagnetic one, it is portable and cheap. As with the electromagnetic mocap system, it has a lot of wires connecting it to the host computer; thus, seriously limiting the performer. The main problem, however, is that the electromechanical mocap suit allows only so many degrees of freedom captured. This is why it is suitable only for simple motion capture tasks. Every major computer graphics exhibition features pretty girls dancing in electromechanical mocap suits. The simple dance movements are pretty much all that these suits can capture.

Optical mocap: Optical motion capture is the most complex and expensive mocap method among its peers. The performer wears a set of *markers* - small balls covered with a highly reflective substance. The positions on the performer's body, where markers should be attached, have to be set very precisely; otherwise, the motion data may be incorrect. When the mocap session starts, the performer starts moving and at the same moment several cameras on a ramp above the scene start capturing frames with high frequency (typically 120 Hz). After the mocap session ends,

images captured by the cameras are submitted to the postprocessing stage where 3D Cartesian coordinates of markers are determined, marker trajectories lost due to occlusions are manually recovered and angular data for the model is derived from the marker positions.

The two major advantages of optical mocap are: **a)** a very large capturing volume, **b)** unsurpassed quality and precision of captured motion data. The large volume is possible because there are no wires connecting equipment on the performer's body to the host computer, and cameras on the ramp control the large area beneath the ramp. The motion capture volume in optical systems is measured by hundreds of cubic meters, while electromagnetic mocap cannot handle more than 30-40 cubic meters of volume. The marker coordinates precision and the overall quality of motion data stems from the high frequency of frame capture, the high resolution of images captured by cameras and the calibration of cameras before each mocap session.

Besides the high price and long data postprocessing stage, motion capture has no major drawbacks. It is system of choice for capturing motion for video games (like FIFA 2001 [6] or NHL 2001 [7]) and for the movie industry.

Mocap is mostly oriented towards capturing the motion of the human body as a whole. However, there are experimental systems doing *facial mocap* - capture of the motion of the human face muscles. Williams attempted to capture face expression, with reassuring results [100]. Judging by his experience and current trends in mocap research labs,⁴ facial motion capture has a future, and the way it is done today is not too far from the idea proposed by Williams.

What is interesting about mocap is that in theory it can serve as an ideal motion synthesis method, if one does not have to take the price into account and if one is limited to *natural* human/animal motion. Price can be coped with (see above), but the static unalterable nature of the data produced by a mocap studio makes it virtually impossible to use those data in projects which require nonrealistic motion. Projects where significant changes in the motion might be necessary are bound to keyframing as well. This is why the issue of editing/transforming/retargeting motion obtained via mocap is of crucial importance for many animation and mocap studios. Ability to edit mocap data keeping its clean and realistic look would save money both for mocap studios and for mocap data consumers. See [62] for an excellent introduction to mocap and its advantages and drawbacks.

2.1.5 Robotics

Some concepts are common for both computer animation and robotics. For instance, inverse kinematics (IK, see [67]) is used in both. Robot designers employ IK to find proper joint angles and the servomotor controlling signals necessary to place a point on the robot's arm in a certain 3D space location. 3D animators can set the trajectory for a point on the animated character's body and IK is used to compute the joint angles necessary

⁴The author has signed an NDA with his corporate sponsor and cannot elaborate further.

for this point to follow the predefined trajectory. IK may be not the best example of the motion synthesis method in our context because it is strictly auxiliary technique serving as a convenient tool for robot programming and 3D computer animation tasks.

Applicability of certain motion synthesis methods to both robotics and computer animation is not a coincidence. Both disciplines deal with animated objects, the difference is that robotics animates real world objects while computer animation deals with drawn objects. If we look into robotics research related to motion planning and control we will see the same concepts that are used in computer animation. There are numerous examples of this. Consider, for instance, the work of Koga et al. [50]. This is pure path planning research applied to virtual 3D characters instead of the robots in the real world.

Another example is the application of the sensor and actuator concepts in the 3D animation area. We can equip robots with sensors to perceive the environment and actuators to move in this environment. Drawing an analogy with humans, sensors play the role of human sense organs and actuators serve as muscles. In [93] van de Panne tried this approach in his set of virtual robots - moving creatures, equipped with sensors and actuators. Figure 2.4 depicts “bouncer” creature, designed by van de Panne. Various types of sensors and actuators used in his system can be seen on figures 2.5 and 2.6.

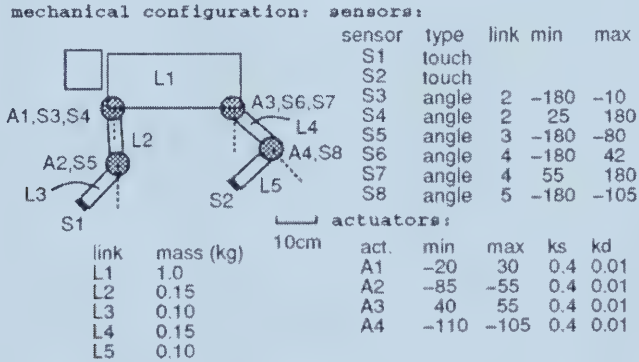


Figure 2.4: The bouncer, one of the virtual robots by van de Panne

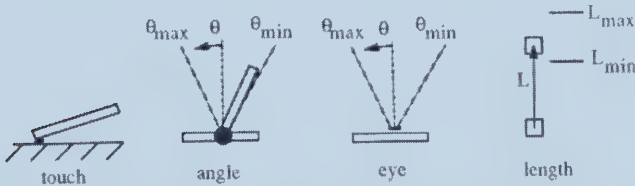


Figure 2.5: Types of sensors used by van de Panne in the SAN project

Numerical simulation of such robots' motion revealed that after necessary tuning of the network connecting sensors with actuators they were able to move in plausible and sometimes even unexpected and amusing ways. Another candidate from the area of

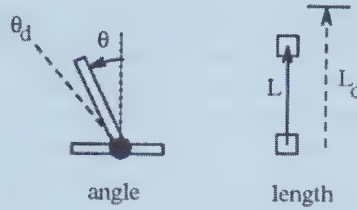


Figure 2.6: Types of actuators used by van de Panne in the SAN project

robotics to be applied to the area of 3D computer animation is *path planning*. The objective of this area of research is to find the optimal motion trajectory for a robot or a robot's extremity given the initial and destination points and obstacles to avoid. This problem does not yet have a general formal solution. Until now the prevailing approach has been heuristic search. Papers by Koga et al. [50], Rijpkema and Girard [82] and van de Panne, Fiume and Vranesic [94] are all applying robotics concepts to computer graphics. Two papers (Koga and Rijpkema) offer solutions for the task of animating a hand and an arm. The latter, by van de Panne, deals with the application of the motion controllers concept from robotics research. As the reader may have noticed in section 2.1.3.1, motion controllers are crucial for motion synthesis via dynamic simulation.

We can see that the final goal of the motion research in computer animation and robotics is essentially the same. Synthesizing the best possible motion trajectories is the ultimate goal for both robotics and computer animation researchers. The difference in the nature of the objects being animated is not that important here. What is important, however, it is the similar complexity and toughness of the problem to solve. Robotics, as well as computer animation, has yet to offer the ideal method for motion synthesis. The desirable characteristics of such a method would be its ability to produce high quality, smooth motion and the availability of high-level control over the process of motion synthesis. If animation for computer games and/or the movie industry is a requirement, motion *naturalness* is important as well.

The main difference between common 3D computer animation and robotics is the concept of feedback and local planning. While in robotics motion of the robots is planned depending on the data the robot gathers from its sensors, the common 3D animation is much more deterministic in this sense. The keyframe animation, for example, strictly defines all the character's degrees of freedom for the whole animation time interval. There is no such process as gathering data about the environment and altering DOFs accordingly.

2.1.6 Biomechanics

Biomechanics has played very important role from the very first attempts to build forward dynamics-based animation systems. There is no better source of information about internal mechanics of a living body than biomechanics. For example, to create proper motion controllers one must know muscle mechanics, to assess simulation results one must get some real life biomechanics data to compare, and so on.

Biomechanics studies the motion of the living organisms, mainly humans and animals.

Therefore, any serious researcher doing experimental animation systems based on the idea of dynamic simulation has to consult biomechanics literature to extract the information necessary for building correct mathematical models of the human/animal locomotion system. Hodgins used biomechanics research results extensively in her human locomotion study [80], [48]. Miller used biomechanics data on the snakes/worms locomotion in [64]. Lee et al. used a biomechanical profile of the facial muscle system to build their realistic facial animation system in [54]. Wei, Zhao, Badler and Lee made an experimental lifting motion animation system, using the results of biomechanics research on the motion of human arms [53]. In fact, every single project involving forward dynamic simulation of motion uses basic biomechanics data - *mass distribution*. Without such data it is impossible to assign correct masses to the character's body segments.

Accompanying biomechanics, the anatomy data is often used in dynamic simulation systems. In the facial animation system by Lee [54] as well as in the realistic human body rendering project by Sheepers [85] knowledge of the human muscular system was crucial. Work by Wilhelms et al. [99] uses a fair amount of animal skeleton anatomy data, to deliver highly realistic rendering of the animal bodies with realistic skin and visible muscle shape change during motion.

Good treatment of biomechanics and human motion control is given in [101], with mathematical details and definitions, that could be useful in mathematical modeling of motion. Reference by Nordin and Frankel [69] is more oriented towards anatomy and can be of interest only to those who are interested in detailed anatomical and physiological data regarding the biomechanics of the human body.

2.1.7 Other research

We conclude the motion synthesis section with a review of the several papers related to motion synthesis which not wholly fit into one of our classification categories. These papers either consider research fitting in several categories at once (for example, a paper combining artificial learning algorithms with dynamics) or are somehow related to one of the categories (rigid body collision detection methods are closely related to dynamic simulation).

Collision detection and response: It would be fair to say that a major part of the modern sophisticated forward dynamic simulation systems is the collision detection and response code. In the virtual world of moving rigid and flexible bodies of various shapes and material properties, detecting collisions and handling them is not a trivial task. Methods for collision detection and response for curved surfaces and flexible bodies are computationally intensive and this is an area of active research these days.

One of the most prominent scientists studying complex cases of collision detection is David Baraff. He has numerous works on collision detection and the calculation of contact forces. One of his earliest papers from SIGGRAPH'89 [11] presents an analytical method for calculating forces for rigid bodies in resting contact.⁵ The

⁵Analytical methods such as Baraff's are not easy to derive. However, when derived and proved

fast collision detection algorithm in [12] exploits geometric coherence of the rigid bodies being animated and uses information from previous timesteps to determine if collision has happened in the current timestep. In [13] Baraff describes his method of simulation of rigid bodies contacting with friction. The joint work of Baraff and Witkin [15] is the first Baraff's paper regarding forward dynamics of non-rigid bodies, unifying previous work on flexible and rigid body dynamics. [14] deals with the task of the fastest possible calculation of the contact forces in a system of colliding rigid bodies.

Among other interesting papers in the area of collision detection we can mention the work of Moore and Wilhelms [66], one of the earlier works proposing algorithms for non-trivial collision cases. It presents two methods for collision detection, one for triangle-approximated surfaces and the other for polyhedral bodies. There are also two collision response algorithms: a simple and general spring model and a faster, more precise, analytical technique applicable to the rigid bodies only.

Herzen et al. developed an algorithm for geometric (without forces) collision detection for parametric surfaces [46]. Gascuel [37] proposed a new formulation for non-rigid body dynamics. In his dynamic simulation system bodies being animated are represented as rigid cores covered with a "soft" layer. This original formulation allows for faster simulation and precise calculation of contact surfaces.

Artificial motion learning: Grzeszczuk and Terzopoulos [44] developed virtual creatures capable of learning their own ways of locomotion. As we mentioned before, in the dynamic simulation of animal/human locomotion the toughest problem is creating proper motion controllers. Grzeszczuk and Terzopoulos shifted the burden of finding an optimal motion controller from the animator to computer. Their learning algorithm consists of two phases - learning low-level controllers and learning high-level controllers. In both phases randomly generated controllers are passed through the optimization engine where the selection of the optimal set of controllers happens. Their results looked quite impressive with artificial snakes, rays and sharks moving gracefully and even performing SeaWorld tricks.

Artificial evolution is probably the better term describing the core of work by Sims [86]. In that paper the author pushes the idea of van de Panne's SAN [93] even further. Unlike the virtual robots of van de Panne, Sims' virtual creatures are created by computer program, simulating evolution in nature. They learn to locomote in the process. Depending on the task set by the author, his system produced various kinds of creatures optimized for swimming, walking and jumping.

Combining kinematic and dynamic animation techniques: Chadwick et al. developed a 3D animation system, called "Critic" [28], which uses layered abstraction of the character model. The inner core, the character's skeleton, is a hierarchy of rigid links, and is animated by keyframing. The second layer is flesh (muscles and fat). The flesh layer employs a complex dynamic model of simulation, thus

these methods produce precise numerical values and are generally faster than corresponding numerical methods.

achieving a fairly realistic look in the process of motion. Combining a kinematic presentation for the skeleton and a dynamics-based flesh allows for an easier learning curve for those animators who would be bold enough to switch from traditional 3D keyframing packages to the “Critter”.

Specific types of animation: Some researchers have focused on very narrow areas in animation. For example, such a specific topic as skin deformation during human grasping was modeled by Gourret et al. [41] It employs detailed physical models of the human hand and of soft objects being grasped. All the necessary material properties of the human skin and objects were included as well, resulting in very convincing animation. We have already mentioned facial animation research by Lee [54], lifting motion simulations by Wei et al. [53] and a general treatment of human grasping by Rijpkema and Girard [82].

2.2 Overview of the motion transformation methods

Synthesis of motion by itself is an area of specific academic interest. In the video games industry the problem of convenient and effective *editing* of motion is no less important if the motion was captured. The problem of motion reuse becomes one of economics. If the character in the game is fully animated and ready for inclusion in the production version of the game, but some cosmetic changes in motion can improve its look, the production team is currently bound to the full recapture of the necessary motion fragments. This option is not economically feasible and game teams are forced to ship the game “as is”.

Experienced 3D animators will smile and produce a speech about the virtues of keyframe animation and the simplicity of editing keyframed motion. However, keyframing is too low-level a technique to be considered for inclusion in games involving a large amount of human motion. Any reasonably realistic sports simulator like FIFA 2001 is impossible to create without motion capture. On the other hand, if the motion was captured, it does not have any “control knobs” like keyframing or even sophisticated forward dynamics-based animation systems. The ideal solution would be to marry the control of keyframing with the realism of forward dynamics. We are still to achieve this level of control over captured motion, but there are other experimental methods which given enough polish (such as a friendly user interface) could be used in the today’s mocap studios.

The most promising motion transformation technique (around which this thesis is built) is the one developed by Zoran Popović. It is described in detail in [78] and briefly in [79]. Other, simpler, techniques of motion transformation existed before Popović. They are described in the works of Bruderlin and Williams [26], Phillips and Badler [71], Unuma et al. [92], and Witkin and Popović [104].

2.2.1 Motion transformation without dynamics

Motion transformation without dynamics is a family of methods which appeared first and are able to handle light, cosmetic changes to the motion. Their advantage is the simplicity

and speed. Warping motion trajectories and applying DSP techniques to them is an easy task for the modern computer hardware. Motion warping is one kind of transformation without dynamics; motion DSP is another. Warping is described in [104] and is used in the quadrupedal locomotion warp method by Samson [84]. Motion DSP (or motion signal processing) is described in [26], with a similar work by Unuma [92]. Yet another IK-based motion transformation is used in the work of Phillips and Badler [71].

Here we present more detailed description of those motion transformation methods

2.2.1.1 Motion warping

[104] is an independent work of Popović on the motion transformation issue. His approach turned out to be unsuitable for serious motion changes involving dynamics (the human jump, for example). The reason is that his representation of motion lacks dynamics, being essentially equivalent to keyframing i.e. purely kinematic. This technique preserves the high frequency fraction of the motion, one that adds a realistic look to it, while allowing animator to add slight variations to the low frequency components. This method fails when more than subtle changes to the motion are required. This paper is not directly related to spacetime optimization, but some ideas were reused by Popović in his Ph.D. thesis [78].

Samson successfully applied motion warping to his quadrupedal gait retargeting system [84]. His experiments with the animated model of the marten demonstrated that the motion warping technique can be applied to the models other than human.

2.2.1.2 Motion signal processing

Work by Bruderlin and Williams [26] applies digital signal processing (DSP) methods to the area where they have never been applied before - 3D animation. That paper gives a wider interpretation of the motion DSP concept than the motion warping of Popović. Bruderlin and Williams employed multiresolution filtering to achieve an exaggerated, smoothed or jerky appearance of the same animation sequence. Multitarget interpolation resulted in the ability to blend various types of motion - blending an “angry walk” with arm-swinging motion would result in motion resembling an angry swinging of arms while walking. Motion waveshaping restricts or maps the motion trajectories in a way similar to waveshaping in digital signal processing. This allows for another means of motion style adjustment. Finally, motion displacement mapping is a way of changing motion similar to keyframing. The animator may adjust character pose at certain frames, and the system recalculates motion curves accordingly.

2.2.1.3 Fourier principles in animation

Unuma, Anjyo and Takeuchi made their own version of a kinematic motion editing engine [92]. Using an approach similar to that of Bruderlin and Williams, they presented the motion data as the rescaled Fourier functional model. With this model it was easy to extract style from one animation sequence and apply it to another one. They added mood to the animated sequence of human run and turned it into a sad (depressed) walk, a tired walk and a brisk walk.

2.2.1.4 Interactive behavior adjustment

Phillips and Badler designed the 3D animation package called “Jack” [71]. In this package, animation of the human body is handled by one of the IK methods. The animator is able to interactively change some constraints on the model such as foot placement, torso bend, center of mass position and pelvis rotation. The human model follows the constraints, using a fast IK algorithm. Since no dynamics were used in this approach to motion editing, we can rank it among the other non-dynamic methods; although, it does not employ any signal processing technique similar to that of the methods by Popović or Bruderlin/Williams. As all the other kinematic methods, this technique can only handle slight changes to the motion. The Phillips/Badler system was designed for motion *synthesis* in the first place, but it fits in our gallery of non-dynamic motion transformation methods, as we can treat the process of adjusting constraints as motion *transformation*.

2.2.2 Motion transformation with dynamics

Kinematic and signal processing-based methods of motion transformation are very appealing in cases where only a slight change of motion is necessary. Once an animator wants to turn the human jump into a jump with a somersault, non-dynamic methods become useless since in this case the dynamics of motion play a crucial role in the scene being edited. Transforming motion with the help of a simulation of Newton’s laws of mechanics would help to achieve extremely realistic looking results in this case.

The method which would allow an animator to introduce drastic changes into animation sequences is called *spacetime optimization* (introduced in section 2.1.3.2), and the underlying concept of *spacetime constraints* (or **SC**) was introduced by Witkin and Kass in [103]. This paper was followed by numerous works of other scientists applying it to various areas of physically based 3D animation. Here we present an overview of those works.

2.2.2.1 Early work of Witkin et al.

[102] is an earlier work of Witkin et al. which focused on constrained methods of animation. In the method described by the authors energy constraints were applied to the parameterized model. An instance of such a model is a system of several objects in space - some machinery detail (joint or valve) or just objects floating in space without gravitation. The paper introduces an elegant mathematical model for parameterized models and proposes a method of the numerical simulation of motion inside such a model. For example, given proper initial constraints and objective function, the model can self-assemble. In another example, the model can react to the external forces and reconfigure itself in attempt to stabilize. It looks like the system has springs inside which return it to the initial configuration after each attempt to break it. However, motion transformation is not the main subject of this paper.

2.2.2.1.1 Spacetime constraints: [103] is the first SC paper touching on artificial motion generation. Partially based on [102], this paper describes a method for motion

generation using the SC concept. These two papers slightly differ. What was energy constraints in [102] becomes an objective function in [103] and constraints acquire a new meaning. Now they span two dimensions, time and space, thus covering the whole character motion domain. Except for that concept shift the papers are quite close to each other. The latter one transfers concepts of the former into the motion construction and processing area but the core mathematical framework was obviously developed by Witkin before 1987.

2.2.2.1.2 Spacetime windows: The paper by Cohen [30] presents the improved implementation of the Witkin/Kass idea. However, it uses different from that of Popović's approach for dealing with numerical complexity when the non-trivial models (human body, for example) are involved. The paper elaborates on one of the first large-scale applications of SC. It is more interesting from the point of view of speeding SC optimization up. Cohen describes his implementation of the SC motion generation and editing engine. One thing that is of particular interest to us is the spacetime window concept. According to it, an animator should split the whole spacetime of the animation to be created/edited into several windows. Each window is essentially a portion of the model DOFs and/or the animation time interval.

By splitting the animation spacetime into windows two goals are achieved. First, a portion of the spacetime is more easily converged to the solution by the SC optimizer than the whole motion spacetime. This is the most important point for us as we are concerned about fast, preferably real time, application of this technique in the area of video games. Second, each window is independent of others so the animator can manipulate the animation sequence by segments, which sometimes may be absolutely necessary. The cat and mouse example from this paper demonstrates the case where the animation sequence must be split into several smaller "windows" in order to make it editable. The spacetime window concept is an alluring idea because it could possibly help bring down the optimal trajectories calculation time.

2.2.2.1.3 Motion blending with spacetime: Several Microsoft researchers presented the fast motion blender at SIGGRAPH 96. Their work is described in [83]. A part of their research was building a fast spacetime constraints optimizer to produce motion transitions in real time. Unlike Popović and Witkin, they opted to use another nonlinear programming algorithm instead of SQP. They chose an alternative dynamics formulation by Balafoutis [10]. By combining Balafoutis formulation and BFGS optimizer they achieved impressive results. Their optimizer works in a range between 20-70 seconds for a moderate complexity motion and that was achieved on a slow Pentium 100 MHz processor for the problem of approximately the same size as the one used by Popović. Compared with a 1-2 minute range on a high-performance SGI Octane for the SQP optimizer from Popović's thesis [78] it shows that the formulation of Rose et al. is worth exploring if one wants to bring the spacetime optimization speed up as much as possible.

2.2.2.1.4 Hierarchical spacetime control: Yet another way of implementing the SC optimizer is described in the work by Liu et al. [58]. This work focuses on the

trajectory basis functions. While Witkin and others used cubic B-splines as the basis for the motion trajectory functions, Liu et al. decided to use wavelets and hierarchical B-splines as a basis. Both the wavelets' and hierarchical B-splines' crucial feature is their ability to change resolution where necessary. In other words, where the trajectory function is smooth or "wide enough" the wavelet basis can have a low resolution in time, leading to shorter iterations calculation time. While in the spacetime areas with disturbed, "narrow" function graphs, basis resolution is increased to reflect those non-smooth, high-frequency details. According to the calculation timing graphs in the paper, the adaptive wavelet basis (adaptive in the sense that the basis resolution is changed automatically depending on the smoothness of the trajectory graphs) has a significant advantage over standard B-spline basis. It has about an order of magnitude less time requirements than the B-spline basis for the SC optimizer to converge to the solution. It seems that adaptive wavelet basis is the most promising idea in the quest to bring SC optimizer execution time down.

2.2.2.2 Other interesting works

Among other applications of the SC optimization concept are two papers by Brotman and Netravali [24] and by Ngo and Marks [68].

2.2.2.2.1 Global search algorithm in spacetime optimization context: Ngo and Marks [68] transformed the original SC numerical optimizer fairly significantly. In their paper they explain the alternative concept of searching the optimal trajectory of motion. Their idea is to apply genetic search technique and evolution simulation methods to the task of finding the optimal motion trajectory. For this approach to work, they changed the internal model representation to a more "biological" way, replacing classic muscle and joint DOFs with stimulus/response (SR) representation. Such a mix of evolution modeling and SR model representation requires massively parallel computers and a lot of time to find an optimal path, but it is able to produce paths impossible to find with classic SC methods.

2.2.2.2.2 Interpolating between key frames with spacetime optimizer: [24] describes the application of the SC concept to the task of inbetweening, or motion interpolation. Previously, motion interpolation was done with cubic splines, adding necessary smoothness to the motion, but such a smooth motion does not always looks natural. There are additional criteria for the motion optimality in the real world, not only smoothness. Brotman and Netravali hypothesized that the additional criterion has to be the energy consumption during the motion of an object. They added several necessary input parameters (mass, gravity, wind, etc.) to the motion interpolation engine and also changed it to include dynamics into calculations. The resulting motion turned out to look much more convincing than the motion done without considering dynamics, with plain cubic spline interpolation techniques.

2.2.2.3 Recent work by Popović

In [79] and [78] the new approach to the motion editing problem has been proposed. The basic idea was proposed by Witkin and Kass in [103], but the spacetime numerical solver used by Witkin was not suitable for the large and complex models, like a human body. Popović designed a workaround for this spacetime method deficiency by introducing two additional steps in the Witkin/Kass algorithm. These are model simplification and reconstruction. Simplification precedes the motion fitting stage (where the model motion is computed) and the motion reconstruction adds original detail level to the transformed motion, keeping its style and feel intact. Model simplification overcomes the major deficiency of the SC method. It allows very complex models to be processed.

Human and animal bodies as well as more sophisticated models became suitable for the SC motion editing. The SC method proposed in [78] and [79] combines the best features of several Witkin and Popovic's works published earlier, including [102], [103] and [104]. Popovic's thesis introduced the notion of motion library. An animator is able to generate an infinite amount of variations of the same motion; in other words, he or she now has a large virtual motion library built from just one motion clip. In the Popovic's SC algorithm the transformed motion looks as realistic as it was before transformation. The motion warping algorithm from [104] became a part of the motion reconstruction process.

The physically based motion transformation system by Popović can be applied for both the drastic and light motion changes. However, it is rather overkill for minor motion editing; it is just too powerful for that. Any motion warping/DSP technique proposed in [104], [26] or [92] can do the same much faster and probably with higher quality of the resulting animation.

2.3 Scope of this thesis

The focus of our research was one of the recent implementations of the spacetime optimization based motion transformation system. The system we worked with was implemented by Zoran Popović; it is the first spacetime optimization based motion transformation engine able to work with very complex models [78], [79]. By employing motion mapping techniques, also based on the optimal control, Popović's system can easily handle motion captured from a human performer. It is based on a paper by Witkin [103] that introduced the idea of transforming motion by optimizing motion trajectory subject to user defined constraints.

In the course of our research we looked into the possibilities of application of this concept to the domain of video games. The ultimate goal was to make it applicable in video games as a real time motion transformation engine. We looked into areas where speed improvements can be achieved. The key area turned out to be the nonlinear programming solver, but it was the hardest part to speed up. We found a way to significantly speed up the motion upmapping module responsible for the motion reconstruction from the simplified form. As our implementation of the upmapping module can do its task in real time, we achieved another goal. We made it possible for the game to store its motion

data in compressed form; thus, significantly reducing disk and memory space occupied by the game as well as decreasing the loading time.

Chapter 3

Physically based motion transformation system

This section is a description of the motion transformation system which was the subject of the research performed by the author of this thesis. We will introduce the reader to the system's user interface, capabilities and internal structure. In the course of covering the system's capabilities, we will elaborate on the three stage motion transformation method used by the system.

We begin our review with introductory information on mathematical programming and on the mathematical program solver, SNOPT, which serves as a transformation engine in the system.

3.1 Principles of the physically based motion transformation

Before introducing the reader to the details of the MTS, the motion transformation system by Zoran Popović, we have to explain some basic mathematical concepts the MTS is built on. Spacetime optimization, first introduced by Witkin [103], is the foundation of MTS. The spacetime optimization itself is a special case of nonlinear programming. Nonlinear programming in its turn is an area of the discipline called *mathematical programming*. See appendix B for introduction into mathematical programming and nonlinear program solver SNOPT, employed by the MTS. It will help the reader to link two such distant concepts as the motion of articulated figures and nonlinear programming. For details on SNOPT and its interface see the SNOPT user's guide [39].

In this chapter we will show how these two concepts can be used together to build the physically based motion transformation system.

3.1.1 Problem setup

The major part of the MTS code deals with converting motion from the input file format to the form suitable for submitting to SNOPT. Input motion data is represented as a set of DOF (degree of freedom) arrays. Sometimes DOF arrays are also called *animation*

channels (see section 1.5). The data accepted by SNOPT includes: nonlinear constraint and an objective function coefficient matrix; arrays with upper and lower bounds; and the user-supplied function pointers defining callbacks for SNOPT to use when nonlinear components of the objective function or the nonlinear constraint function has to be calculated during the solution process. By definition, the process of converting motion representation from the DOF array to the NLP format cannot be done without user intervention. The user must define *what exactly* he/she wants from the system by setting up a proper objective function and constraints.

This process involves two very different activities.

First, the user must edit the source code and set up the primary criterion of motion transformation - the objective function. The objective function tells the system what is the landmark which SNOPT will use when searching for the optimum, i.e. for the best motion trajectories. The other major part of the problem which must be set up are the constraints. This is also done by editing the source code. The user expresses the features wanted in the resulting motion by setting the necessary constraints. Both the objective function and the constraints are expressed in the terms of C++ classes taken from the internal class library which is the part of the MTS source code.

Second, the user might adjust some problem properties through the system's user interface (UI). Such properties as gravity, the weights of the objective components, the state of the model DOFs at the beginning and at the end of motion, and others, can be adjusted through the UI.

When the user has coded the problem by setting the objective function and the constraints in the source code, and when he/she has adjusted all the properties he/she wanted through the UI, the nonlinear program is formulated and is ready to be submitted to SNOPT for solution.

The idea of presenting motion as a nonlinear program was first proposed by Witkin and Kass [103]. Numerous other systems beside the MTS by Popović use the same underlying concept as was proposed by Witkin and Kass. Motion transformation systems by Cohen [30] and by Rose et al. [83] all employ that idea.

In the next few sections we will describe in detail the process of setting up the nonlinear program based on the input motion data presented as a set of DOF arrays.

3.2 Implementation of the physically based motion transformation system

This section explains the internals of the MTS. It was conceived as a hybrid between the user's guide and the developer's overview of the system internals. It has material interesting for both users of the MTS and developers who might want to add some new functionality to the system.

From the developer's point of view, the MTS by Popović consists of 31 modules written in C++ and 7 modules written in C. The C modules contain the utility functions for working with dynamic arrays (`array.c`) plus an assortment of math functions operating quaternions, vectors and 4x4 matrices (all the other C modules). The C++ modules are

described in the table C.1, appendix C.

Figure 3.1 shows the global distribution of the MTS modules into five categories - core, I/O, UI, utils and SNOPT. Note that not all modules from the table C.1 are in this diagram. Only the most important are presented there.

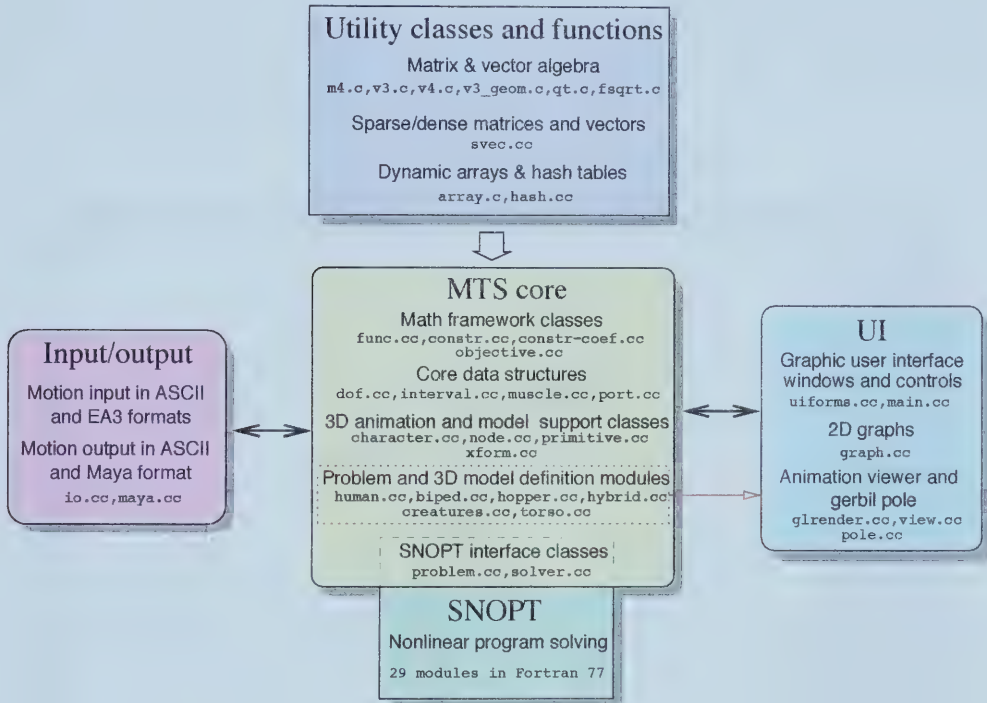


Figure 3.1: All the modules in the system, distributed into five categories

3.2.1 User interface

The most important part of the system's user interface is the animation viewer widget. It takes the largest part of the whole main window area and has roughly 40% of the UI code dedicated or related to it in some way. The second important (and tough to implement) part of UI is the 2D graph window. It has a convenient interface allowing the user to adjust the scale and shift graphs along X and Y axes, as well as performing numerous operations on graphs, such as smoothing them or flattening their ends.

Aside from these two important parts, everything else is just different incarnations of the forms built with the XForms package [106]. A form built by the XForms UI designer combined with automatically generated *callback stubs* constitutes a ready-to-use user interface skeleton that requires only filling stubs with an application logic. This is exactly the way the simpler parts of the UI in the MTS were done. The system's source code is accompanied by several *form definition files*. When the system is being built from the

source code, those files are processed by the XForms' UI designer. The result of this process is the source code file with callback stubs. The module `uiforms.cc` is such a template filled with code providing necessary functionality for the MTS.

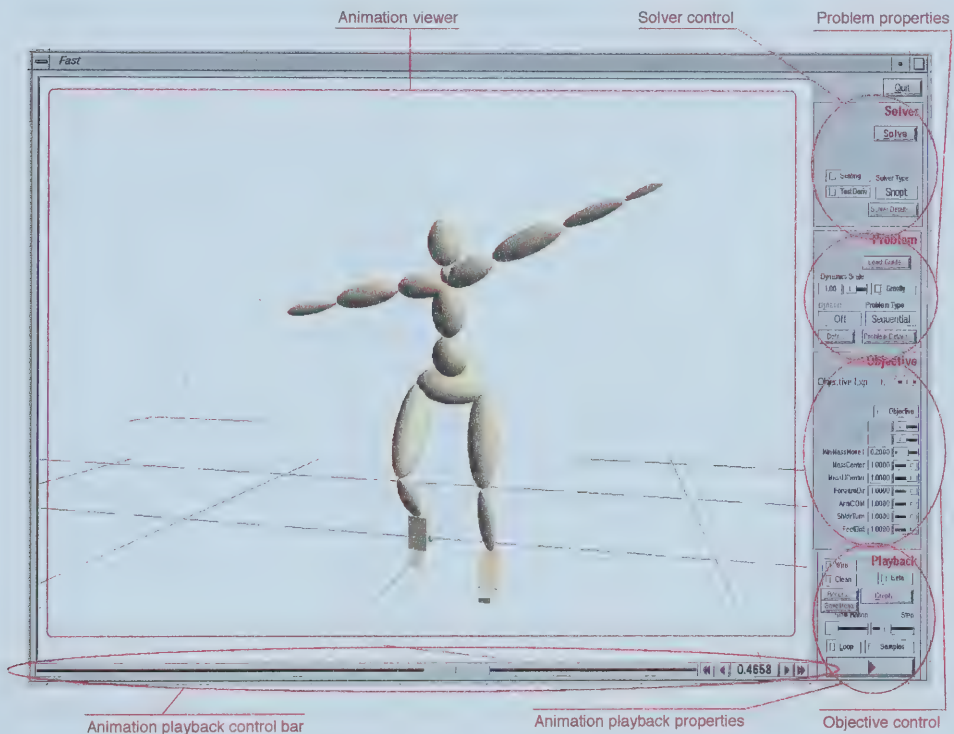


Figure 3.2: Screenshot of the main window with its panes and controls identified

We will start detailed description of the various UI elements with the animation viewer and continue with the animation playback controls and secondary windows.

3.2.1.1 Animation viewer

The animation viewer takes the largest area inside the main window, approximately 90% of the total window area. The animation viewer is a window into the virtual world where the 3D animated character moves. The user can rotate, zoom and pan the camera around the scene with the 3D character model. The primary purpose of the animation viewer is to allow the user to visually inspect the results of the motion transformation. Additionally,

the user can use this viewer to obtain precise 3D coordinates of the pose constraints to be applied to various body parts of a character.

Camera & gerbil pole control: The user can view the scene from any angle and with a wide range of zoom. The other interactive element of the animation viewer is the gerbil pole - a 3D pointer controlled by the 2D positional device, a computer mouse. The actions available to the user and the corresponding keys and/or mouse button combinations are summarized in table 3.1

Working with gerbil pole and pose constraints: The camera control is obvious but the gerbil pole functionality requires additional explanation. The word “gerbil” in its name originated from the older implementation of such a pointer which had a name “mouse pole”. Popović rewrote parts of the code and renamed it the “gerbil pole”. The word “pole” stems from the visual appearance of the pointer because it reminds one of a thin pole with a ball at the tip. However, the most important feature of the gerbil pole is not visible in the animation viewer. One must look at the system’s standard output to understand why the gerbil pole is so crucial for the system functionality. Each time the user moves the gerbil pole, the pole’s 3D coordinates are printed to the standard output.

Let us suppose the user wants to set a constraint on the character’s left foot so that it is “nailed” to the ground during some time interval. By manipulating the gerbil pole, the user can discover precise 3D coordinates of the spot he/she wants the character’s foot to be “nailed” to; then he/she proceeds to edit the source code and to enter those coordinates manually as part of the pose constraint definition. Of course, this kind of user interface is unacceptable for a commercial product to be used in a mocap studio or in an animation studio but it is pretty much sufficient for research purposes.

3.2.1.2 Animation playback control bar and other main window controls

Other elements of the main window include the animation playback control bar, the solver control pane, the problem properties pane, the objective control pane and the animation playback properties pane (see figure 3.2). The description of these UI elements follows.

Animation playback control bar: The user can control animation playback with this control bar. It has a slider, a time indicator and four buttons controlling frame display. The slider can be used to quickly jump to the necessary frame in the animation. The time indicator shows the time corresponding to the current frame. The four buttons on either side of the time indicator allow for a frame-by-frame animation playback and for a jump to the first or the last frame of the animation.

Solver control pane: The solver can be started and stopped here. Besides the solver start/stop button there are additional controls in this pane. The two most interesting among them are the solver chooser and the “Test derivs” checkbox. The solver chooser allows the user to select the nonlinear program solver preferred. The solver of choice is SNOPT but there are interface classes for other solvers in the source

| Key/mouse button | Action |
|---------------------------|--|
| Left mouse button | Click - Depending on whether the spot the user clicked on was located on the character's body, the gerbil pole will either move to the clicked point on the character's body or it will stay where it was, but the mouse pointer will jump to the current pole's location. Drag - The gerbil pole moves in the horizontal plane along with the mouse. Horizontal means from the point of view of the 3D character, as the user can place the camera anywhere and there are no horizontal or vertical planes from the user's viewpoint. |
| Middle mouse button | Click - This is similar to the left mouse button. Drag - This moves the gerbil pole in a vertical plane. Vertical means perpendicular to the horizontal from the point of view of the animated character, but always facing the camera. In other words, the vertical plane always stays parallel to the vertical axis of the character, rotating about it when the user rotates the camera. |
| Right mouse button | Click - This is the same as for the left mouse button. Drag - This does nothing. |
| Shift+Left mouse button | Drag - rotate camera |
| Shift+Middle mouse button | Drag - pan camera |
| Shift+Right mouse button | Drag - zoom camera |
| Ctrl+Left mouse button | Click - This moves the gerbil pole to the point of view. Drag - This slowly moves the gerbil pole in the plane perpendicular to the vector of view. Useful for precise gerbil pole positioning. |
| Ctrl+Middle mouse button | Click - This is the same as for the Ctrl+Left mouse button. Drag - This slowly moves the gerbil pole along the vector of view, i.e. moves it closer or farther away from the camera. Useful for precise gerbil pole positioning. |
| Ctrl+Right mouse button | Click - This is the same as for the Ctrl+Left mouse button. Drag - This is the same as for the Ctrl+Middle mouse button. |

Table 3.1: Functions available in the animation viewer and their activation keys/mouse buttons

code. Hence, it is possible to compile the MTS with several solvers plugged in and to choose between them in run time with this UI element.

The “Test derivs” checkbox was useful for debugging purposes. It would allow Popović to see if function derivatives inside the system were calculated correctly. Apparently, this UI element became obsolete when debugging was over.

Problem properties pane: This pane contains two important controls - the problem type chooser and the dynamic constraints placement chooser. Setting the proper problem type with the appropriate chooser is seminal for getting the motion transformed. The problem types available are: *spacetime*, used during the motion transformation stage; *sequential*, the one for the motion up and downmapping stages; and *spacetime iterative*, a special problem type for solving a problem over a subinterval inside the animation. The dynamic constraint placement chooser determines where the Newtonian constraints will be placed in the problem to be solved. One can either turn them off completely (rendering the transformed motion unrealistic), place them in the objective function or place them in the constraints array. Moving constraints in the objective function might seem to be quite stupid, but it is not. By moving constraints in the objective function, one can impose the same constraints but in a more “mild” manner. The “Elastic” mode in SNOPT does exactly that [39].

Other controls in this pane worth mentioning are the gravity checkbox, which turns gravity on/off in the virtual world, and the “Load guides” button which loads additional motion mapping data called “guides” (see section 3.2.2 for explanation).

Objective control pane: The objective control panel does what its name says - it allows the user to modify the objective function in run time. The objective function components are set in the source code when the motion transformation problem is being set up. The component weights, however, are adjustable through this control pane. One can increase the weight or even set it to zero, eliminating the corresponding objective component altogether.

Animation playback properties pane: In this pane, the user is able to choose between two modes by which the character is displayed in the animation viewer. Also one finds here probably the most important button for the whole system - the Play button. The detailed description of the UI elements inside this pane follows below, in table 3.2.

3.2.1.3 Problem details window

Figure 3.3 depicts the problem details window and its three panes. This window is a secondary one and is called from the main window. The user can adjust various problem-related settings there. We will explain all three panes of this window below.

Problem interval and resolution: This pane deals with the problem interval and the amount of samples allotted to represent it (in other words, the interval resolution). Usually, the problem interval is set to represent the whole problem, from the first

| UI element | Description |
|---------------------------------------|--|
| Wire checkbox | Switch between wireframe and solid modes of drawing 3D primitives of the character's body. |
| Clean checkbox | Toggle display of the additional problem related objects, such as constraints and ports. |
| Beta checkbox | Toggle display of forces. |
| Record button | Record animation as a sequence of PPM (portable pixel map) files, one file per frame. |
| SaveMaya button | Save animation in the Maya 1.5 ASCII format. |
| Graph button | Bring up the 2D graph window (see section 3.2.1.5). |
| Slow Motion & Step sliders | Control the speed and smoothness of the animation playback. |
| Loop checkbox | Make animation loop infinitely or stop after reaching the last frame. |
| Samples checkbox | Choose whether the animation frames will be played only for the moments of time where the DOF samples exist or the interpolated frames will be created, thus making animation look smoother. |
| Play button | Big button with a triangle on it is the main button in the MTS. When the tedious process of setting up the motion transformation problem is over, when SNOPT finished a long process of crunching numbers and found the solution, it is time to press that button and enjoy (or, depending on the results, mourn) the transformed motion of the character. |

Table 3.2: Playback option controls available in the playback properties pane

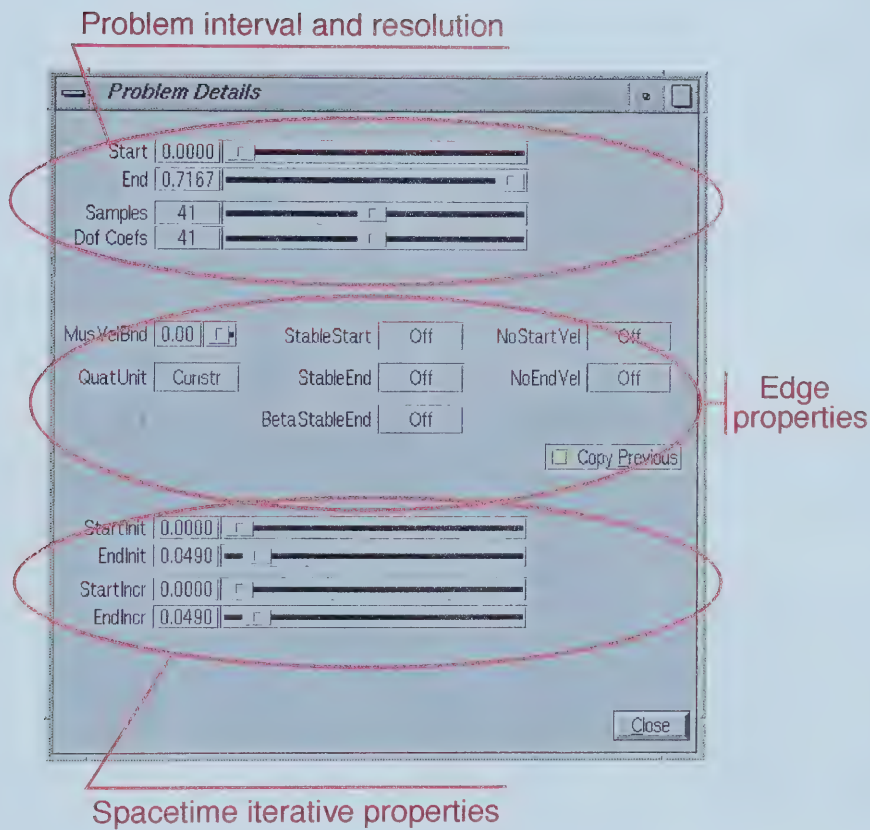


Figure 3.3: Screenshot of the problem details window with its controls identified

animation frame to the last one. It can be changed if the user wants to transform a part of the animation instead of the whole. The two resolution sliders set how many samples will represent the animation, with higher values giving better resolution and smoother motion.

Edge properties: Edge properties are the special settings that pertain only to the first and last frame of the animation. The user can set the initial and/or the final character pose to be stable; that is, to keep the balance. It is also possible to set zero acceleration and zero forces at the beginning or the end of animation. The example of manipulation with the edge properties, where the “stable pose” restriction on the last frame of the animation was lifted, has been demonstrated in the Popović’s SIGGRAPH paper [79] and Ph.D. thesis [78]. Look at the example with the transformed human jump, where the character falls down after landing.

Spacetime iterative properties: This group of controls is in effect when the problem type is set to the *spacetime iterative*. It sets the length of the iterative subproblem to be solved at each step and the moment of time where the spacetime iterative problem begins.

3.2.1.4 DOF details window

It seems that the better name for this window would be the “DOF fixation window”, since the only available action there is fixing DOF coefficients at the first and the last animation frame. Figure 3.4 is a screenshot of an upper part of this window. Additional DOFs cut with the bottom part do not add meaningful information. With them the screenshot would be too tall to fit in page.

Fixing DOFs is necessary whenever the animated character needs the constant pose or extremity position at the beginning or at the end of the animation. “Constant pose” means that all the character’s DOFs are fixed and “fixed extremity position” means that the DOF corresponding to a particular extremity is fixed. For example, if it is necessary for the character to keep the initial position of its left leg unchanged through motion transformation, the user must fix initial DOFs for at least three leg joints. These are the hip joint DOFs, the knee joint DOFs and the ankle joint DOFs. By fixing DOFs for arms, legs, head or torso, the user can achieve full control over the character’s initial and final poses.

Two panes of this window allow the user to fix DOFs either globally or individually. The global pane contains the set of two sliders and a checkbox which set how many edge DOF coefficients must be fixed. One can fix up to 3 edge coefficients; thus, it is possible to fix the initial/final velocity and acceleration as well as position. The same set of two sliders and a checkbox is presented in both the global DOF pane and the individual DOFs pane. The difference is that the global DOF pane controls affect (naturally) all the DOFs of the character, where individual DOF controls affect only their corresponding DOFs.

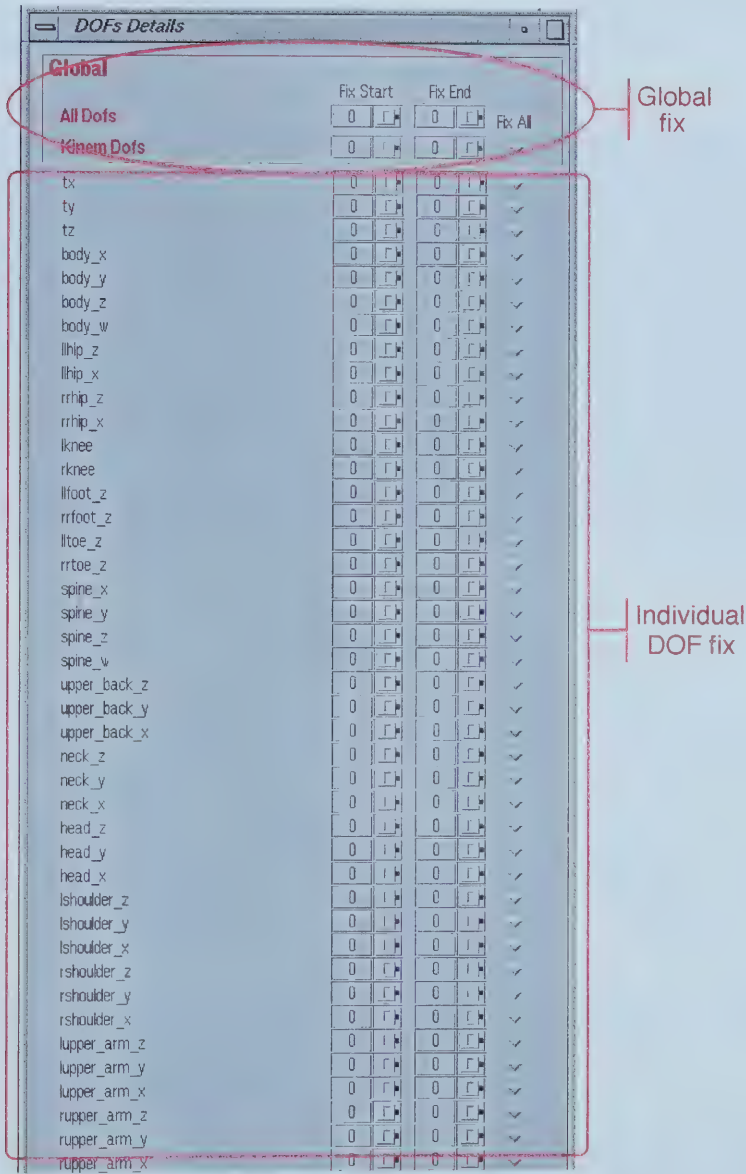


Figure 3.4: Screenshot of the window that should have been called “DOF fixation window”

3.2.1.5 2D graph window

Since in 3D animation the character DOFs are usually scalar functions of time, they can be presented as 2D graphs. The 2D graph window, depicted in figure 3.5, is the second sophisticated part of the system UI after the animation viewer widget.

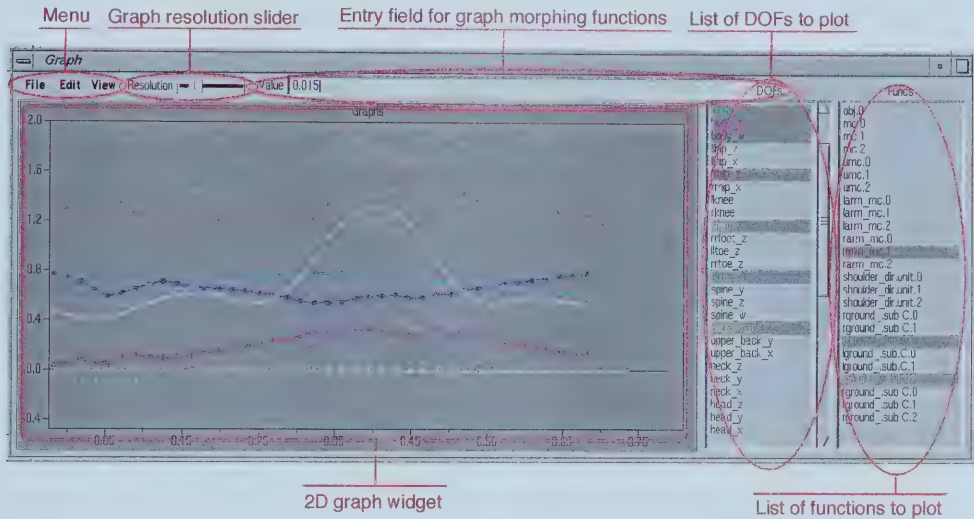


Figure 3.5: Screenshot of the 2D graph window

We will complement figure 3.5 with two tables covering controls and widgets inside the 2D graph window. Table 3.3 explains the 2D graph window elements, such as the 2D graph widget, the resolution slider and others.

Table 3.4 explains key/mouse button combinations allowing the user to zoom and pan the graphs inside the 2D graph widget.

The graph window can display not only the DOF graphs but also the graph of any function made *graphable* in the source code. We will demonstrate the code snippet that creates the function object and makes it graphable.

```
TripleFunc *forward_dir = new TripleUnitFunc(
    new SubTripleFunc("forward.dir", body_forward, body_center),
    TripleUnitFunc::Ydim);
forward_dir->isGraphable(1);
```

`TripleFunc` is a C++ class representing a triple-valued function. `TripleUnitFunc` is its descendant representing functions with vector values of a unit length. In this example, taken from module `human.cc`, the `forward_dir` object is a function whose value is a unit 3D vector representing the orientation of the character's body. The second statement (a call to method `isGraphable()`) is the one that makes the function `forward_dir` graphable. The three components of the function value vector will appear in the displayable function graph list (the **Funcs** list in table 3.3).

| UI element | Description |
|--------------------------|---|
| Menu | The menu includes 3 drop-down menus - File , Edit and View . The File menu allows the user to save the transformed motion in the MTS text file format, which is a plain set of all the character DOFs plus certain problem properties, such as the interval and problem type. The Edit menu has all the graph morphing functions inside. The user can do graph smoothing, edge flattening and other morphing operations. The lower group of 8 operations requires a numerical argument which should be entered via the Value entry field (see its description below). The View menu allows the user to toggle the sample display mode and reset the DOF and the function list selections. The sample display mode determines whether the DOF/function samples are visible on the graph and the list reset function clears the 2D graph widget by deselecting all the DOFs and functions in both lists. |
| Resolution slider | This slider works only when the sample display mode is off. In this case the user may adjust the graph resolution which defines graph smoothness. When the sample display mode is on, the samples determine the graph resolution. It becomes a fixed value and the resolution slider does not affect graphs. |
| Value entry field | See the Menu description above. The value entered here is needed by certain graph morphing functions from the Edit drop-down menu. Note that just entering value here is not enough. One must press the “Enter” key after typing a number there to actually enter it in the system. |
| DOFs list | This is the multiple selection list. The list items are selected by clicking and dragging the left mouse button over the list. As the DOFs in the list are being selected, their graphs immediately appear in the 2D graph widget. |
| Funcs list | The function list has the same interface as the DOF list above. The user has to click and drag the left mouse button over the list to choose the function graph to be drawn. |
| Graphs widget | This is the central place of the graph window, all the graphs are drawn here. The user can control the zoom level and the graph position in the widget. Table 3.4 elaborates on that. |

Table 3.3: Table of the UI controls inside the 2D graph window

| Key/mouse button | Action |
|---------------------------|--|
| Shift+Left mouse button | By dragging the mouse, the user can pan the graphs around. |
| Shift+Middle mouse button | This key combination allows one to change the vertical and horizontal zoom (in other words, scale) level. To change the horizontal scale the user must drag the mouse left or right. To change the vertical scale the user must drag the mouse up or down. |

Table 3.4: List of actions the user can perform on graphs inside the 2D graph widget

Aside from the absolutely necessary functionality of saving transformed motion in the output file, the 2D graph window comes to the rescue when SNOPT stumbles for some reason, and fails to find the nonlinear program solution. Quite often this happens because DOFs are “overfixed” with regard to constraints. In other words, it is possible to define the initial or final character pose that is hard to achieve while keeping the motion physically correct. In this case, the **Funcs** list is the place to look. The user should examine Newton’s constraint graphs, looking for significant deviations from zero. If such a constraint is found, this means the user has to “free” or “unfix” the corresponding DOF in the DOF details window (section 3.2.1.4). This requires the user to be familiar with the source code of the problem/model setup module and the user should not forget to set all the Newton constraints graphable, as shown in the source code example above.

3.2.2 Input/output

The MTS by Popović is a motion *transformation* system, meaning that there are two data streams. One is the original motion data and the other is the transformed motion data. Both are kept in disk files. The original motion data file format supported by the MTS is the ASCII text file containing the basic problem and model properties together with the set of DOF arrays. In the course of research, the author of this thesis has added support for another file format, called EA3 or “EA Cubed” format. It is a motion storage format used by the Motion Capture Studio and Tools and Libraries departments of Electronic Arts Canada¹. After the EA3 format support was added, it became possible to experiment with motion captured in the EA mocap studio. With EA3 import functionality the research was not constrained to the set of data provided by Popović with his system.

Below, we present a more detailed description of the motion data input/output functionality and some technical details on the DOF basis functions used for constructing smooth interpolated DOFs from of a set of samples loaded from the input file.

3.2.2.1 Original MTS motion data file format

The version of the motion transformation system by Popović in its pure, unmodified form supports only one file format. The format is a text file, containing two parts. The first

¹The research sponsor of the author of this thesis.

part is the preamble. The second is the motion data itself and/or other data arrays, such as motion mapping guides. We will talk about guides in section 3.2.3.2.

The preamble looks like this:

```
#Tag=fast-0
/***** defaults start
char* Character::build_name = EAHuman

double Anim::frame = 0.392248
int Anim::show_beta = 0
int Anim::show_clean = 1
int Anim::show_wire = 0
int Anim::frame_delay = 0
int Anim::samples_only = 1
double Anim::step_pct = 0.012600
int Anim::loop = 1
int Anim::playing = 0

Interval *Character::interval = [0.000000, 0.766667, 44]
double Character::gravity[3] = {0.000000, -9.810000, 0.000000}

int Objective::obj_active = 1
double Objective::scale = 1.000000

double NewtonConstr::factor = 1.000000
ProblemType Problem::type = Sequential

RangeInterval *Problem::range = [0.000000, 0.766667, 44]
ConstrCoef::State Problem::quat_unit = Constr
ConstrCoef::State ProblemSpacetime::newton_state = Off
ConstrCoef::State ProblemSpacetime::stable_start = Off
ConstrCoef::State ProblemSpacetime::stable_end = Off
ConstrCoef::State ProblemSpacetime::beta_stable_end = Off
ConstrCoef::State ProblemSpacetime::no_start_vel = Off
ConstrCoef::State ProblemSpacetime::no_end_vel = Off
double ProblemSpacetimeIter::start_init = 0.000000
double ProblemSpacetimeIter::end_init = 0.049000
double ProblemSpacetimeIter::start_incr = 0.000000
double ProblemSpacetimeIter::end_incr = 0.049000
int ProblemSeq::copy_previous = 1

SolverType Solver::type = Snopt
int Solver::test_derivs = 0
int Solver::scaling = 0
```



```
***** defaults end */
```

The first line is the tag line, it must always be in this form. Otherwise, the system will reject the data file as not conforming to the format. After the tag is found the system proceeds to load the problem properties. First, it looks for the line “/***** defaults start”. This line should precede preamble data. Again, if it is not there, the system will not start.

When the preamble data is being loaded, the system expects it to be in a certain order, exactly as shown in the example above. The preamble data is a set of variables. Some of them being real numbers while others are of a boolean type. The names of the variables conveniently follow their names inside the source code. This really helps the novice user of the system as it is a matter of doing a text search once to find where the particular variable is declared in the source code and what are its semantics. As a matter of fact, the MTS author named the variables by copying and pasting their declarations taken from the C++ source code. Hence, the syntax.

Most of the variables in the preamble, with the exception of `Character::build_name` and a couple of others, reflect the state of some user interface control, either in the main window or in one of the secondary windows. Table 3.5 covers the variables and the UI controls whose state they reflect.

| Preamble variable | UI controls | Description |
|---|---|---|
| <code>Anim::frame</code> <code>Anim::show_beta</code> <code>Anim::show_clean</code> <code>Anim::show_wire</code> <code>Anim::frame_delay</code> <code>Anim::samples_only</code> <code>Anim::loop</code> <code>Anim::playing</code> | Animation playback properties pane in the main window | These variables set various properties of animation playback, such as wire-frame or solid rendering, animation playback speed and others. |
| <code>Character::interval</code> <code>Character::gravity</code> | None | The problem interval (i.e. animation time span) and the gravity vector. |
| <code>Objective::obj_active</code> <code>Objective::scale</code> | Objective control pane in the main window | Objective function control. |
| <code>NewtonConstr::factor</code> <code>Problem::type</code> | Problem properties pane in the main window | The problem type and dynamics scale. |
| <i>continued on next page</i> | | |

| Preamble variable | UI controls | Description |
|---|--|---|
| <code>Problem::range</code> <code>Problem::quat_unit</code> <code>ProblemSpacetime::newton_state</code> <code>ProblemSpacetime::stable_start</code> <code>ProblemSpacetime::stable_end</code> <code>ProblemSpacetime::beta_stable_end</code> <code>ProblemSpacetime::no_start_vel</code> <code>ProblemSpacetime::no_end_vel</code> <code>ProblemSpacetimeIter::start_init</code> <code>ProblemSpacetimeIter::end_init</code> <code>ProblemSpacetimeIter::start_incr</code> <code>ProblemSpacetimeIter::end_incr</code> <code>ProblemSeq::copy_previous</code> | The problem details window | The problem interval, edge properties and spacetime iterative interval settings. |
| <code>Solver::type</code> <code>Solver::test_derivs</code> <code>Solver::scaling</code> | Solver control pane in the main window | Selection of the solver and other options. The solver types acceptable for the <code>Solver::type</code> variable are: <code>Snopt</code> , <code>Conopt</code> , <code>Minos</code> , <code>Cfsqp</code> and <code>Rsqp</code> . Not all of these solver names are permitted as the variable values. The rule is that the corresponding solver must be compiled in the system. If it is not, then the file parsing code will detect that the requested solver is not available and will abort the execution. |
| <i>continued on next page</i> | | |

| Preamble variable | UI controls | Description |
|------------------------------------|-------------|---|
| <code>Character::build_name</code> | None | The most important variable, telling the system which model to load. The models are defined in their own C++ modules, see the bottom of table C.1. Acceptable models are: Human, EulerHuman, MocapHuman, RemapHuman, Hopper, FootHopper, Biped, Pogo, EAHuman, RemapEAHuman, Hybrid and Torso. Each model has its own unique characteristics and area of application. Some of them will be covered later, in section 3.2.3. |

Table 3.5: List of the variables from motion file preamble with their corresponding UI controls

The data part follows the preamble. It consists of a series of real number arrays. These numbers are called *samples* or, using terminology by Popović, *coefficients*. The DOF array representing one of the hip DOFs might look like this:

```
double lhip_x_samples [43] = {  
-5.200968e-02, 1.488880e-01, 2.533984e-01,  
. . . . .  
-1.444087e-01, -8.174329e-02, -1.011082e-01, -1.169727e-01  
};
```

For brevity, in this example most of the numbers are omitted. As one can see, the syntax of the array declaration is again C++ like. The striking similarity between the motion file format and the C++ code shortens the learning curve for the novice system user (assuming the user has some experience in C++). One last note about the file format: the length of array must be correct. In our example above, there must be at least 43 real numbers in the array, separated by commas. Otherwise, the input file syntax parser (in module `io.cc`) will fail and the execution will abort.

3.2.2.2 Input data

When the MTS is started, it must be told where to get input motion data. The motion data file names should be specified in the command line after the name of the executable

file. Let us assume that the MTS executable is named `t_fulldds1`, then we can launch the MTS with command like this:

```
t_fulldds1 torso_original torso_airplane
```

This loads two motion data files, `torso_original` and `torso_airplane`. The problem and model properties from the preamble are loaded from the first file only. For all the others, this part of the file is ignored and only the DOF and guide arrays are loaded.

The second form of the command line options includes `EA3` file name as the first option to go after the executable file name, followed by two numbers and the other file names. The numbers specify the frame range to be loaded from the `EA3` file. For example, the following command launches the MTS with three data files, including the primary `EA3` motion data file.

```
t_fulldds1 JMP_BASK_E1.ea3 14 57 hybrid_uniform_dofs torso_airplane
```

Here we load 44 frames of animation from file `JMP_BASK_E1.ea3`, starting from frame 14. We also load a set of guides from two additional files, `hybrid_uniform_dofs` and `torso_airplane`.

The two different forms of command line options are distinguished by the extension of the first file name. If it is “`.ea3`”, then the second form is used. In the second form, the frame range is necessary and everything that follows is optional. The user chooses which form to employ depending on which file format is used to store the motion. The first form is for the original format by Popović. The second one is for `EA3` files. Note that in this case only one file (the first one) can be in the `EA3` format. The files that follow the frame range have to be in the original MTS format by Popović.

3.2.2.3 Output data

It is possible to save the transformed animation in two formats. One is the original MTS format and the other is the Maya text format. To save the transformed data in the MTS format, the user has to call up the 2D graph window with a **Graph** pushbutton in the animation playback properties pane in the main window. Then one has to go to the **File** pull-down menu and choose the **Save** item. Next, the system asks for a file name in order to store the motion data. It then writes the preamble with the current UI, problem and model settings. The preamble is followed by a set of the real number arrays, containing all the DOFs and functions in the system.

To save animation in the Maya 1.5 ASCII text file, the user has to click the **Maya** button in the animation playback properties pane in the main window. Unlike saving in the MTS format, the Maya file will not contain either preamble data or the functions. The only information saved is the model and its DOFs for the whole duration of the animation.

3.2.2.4 Types of DOF basis functions used

As it has been mentioned in section 3.2.2.1, arrays in the data part of an MTS motion file contain function samples, or coefficients, as Popović called them. Obviously, there should be a way to reconstruct smooth functions from those samples. The interpolation is done with basis functions. There are several types of basis functions available in the system. Figure 3.6 is a class inheritance diagram for the DOF classes with class `Dof` descendants differing by the type of basis functions used.

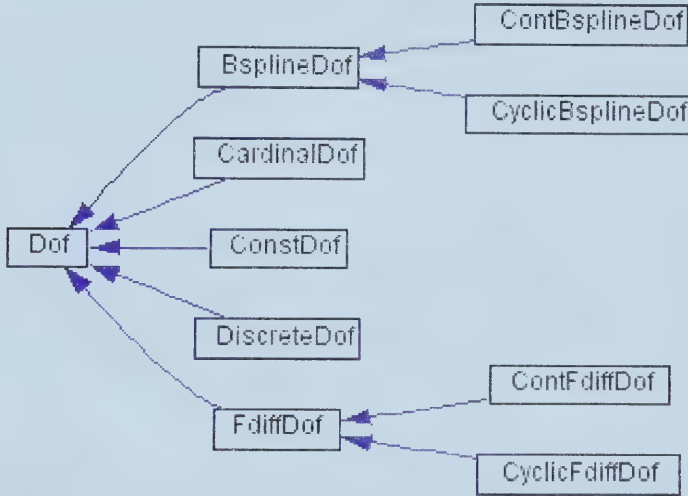


Figure 3.6: Inheritance diagram of the C++ classes representing DOFs

The type of DOF class to be used when coding the motion transformation problem (see sections 3.2.3 and 3.2.3.4) depends on the motion transformation phase. For mapping motion between full and simplified models the forward difference DOF should be used (class `FdiffDof`). For the motion transformation phase the B-spline based DOFs should be used (class `BsplineDof`).

3.2.2.5 Loading animations produced by Electronic Arts motion capture studio

To be able to load any motion produced by the Electronic Arts motion capture studio into the MTS the author of this thesis had to add several routines to module `io.cc`. Here is the source code sample, demonstrating a part of code loading EA3 file.

```

/// load mocap data in EA3 format
/*!

    Note that this function was designed to work specifically with the
    EAHuman model, and won't work with anything else
    \param fname name of the EA3 file to load animation channels from
```



```
\param bf first frame of the animation to load
\param ef last frame of the animation to load
*/
void loadEA3(const char *fname, const short bf, const short ef)
{
    EA3Scene scene;
    scene.readFile(fname);
    EA3Block* anim = EA3Expr::findFirst(&scene,
                                         EA3Expr::HasType("animation"));
    EA3Block* chArray = EA3Expr::findFirst(anim,
                                         EA3Expr::HasType("channel"));
    short nsamples=ef-bf;

    . . . .

    load_ea3_dof(NULL, chArray, "ROOT", 1, bf, ef);
    load_ea3_dof("root", chArray, "ROOT", 0, bf, ef);
    load_ea3_dof("back", chArray, "backBone", 0, bf, ef);
    load_ea3_dof("lhip", chArray, "leftHipBone", 0, bf, ef);
    load_ea3_dof("rhip", chArray, "rightHipBone", 0, bf, ef);

    . . . .
```

The code here uses the EA3 library developed inhouse for the games and tools produced by EA development teams. The EA3 file format is an essentially hierarchical database with certain inclusion rules specifying which containers can include which objects. For instance, the container **Bone** can have only other **Bones** inside. The strict inclusion rules take care of EA3 file integrity and also add proper semantics to the general-purpose database approach employed by the EA3 library. The code above opens the EA3 file, loads container **animation** from there, extracts all the animation **channels** from it and calls the `load_ea3_dof` function (also written by the author of this thesis) for each animation channel extracted. So for the MTS itself the load process ends with the same result as with the loading of the original MTS text file. All the necessary internal structures are initialized; and after the `loadEA3` function returns, the system is ready to play and/or transform motion.

3.2.3 Motion mapping and transformation

After the two previous sections on the system's user interface and file formats, it is time to discuss the core system functionality - the motion transformation.

As Popović mentioned in his SIGGRAPH 99 paper and Ph.D thesis [79] [78], his motion transformation system uses a clever trick allowing it to transform the motion of complex articulated figures quite significantly, keeping it realistic and looking physically plausible. The trick is to map motion from the full model to the simplified model, to change the motion of this simple model and then to apply the changes to the full model.

Thus the full model obtains new motion and still retains certain characteristics of its original motion intact. The balance between which aspects of motion are changed and which are left intact depends on the simplified model. The degree to which the motion is changed is adjustable too; but it does not depend on the choice of simplified model. It depends on the choice of weighting factors in the objective function of the upmapping problem (see section 3.2.3.5).

In the sections below, we will show how the process of the motion transformation is performed on the simplified model. Both motion mapping types, *downmapping* and *upmapping*, will be discussed as well. We will show how the system user can do all this through editing the source code, loading the appropriate set of motion data files and adjusting necessary optimization problem parameters via the user interface.

3.2.3.1 C++ classes representing objective function components and constraints

Before dealing with the intricacies of the correct problem setup we will give two class diagrams presenting a palette of tools available to the user for constructing objective function and constraint definitions. Figure 3.7 presents the inheritance diagram of all the objective component classes available.

When the objective function is created in the code, the object of the class `Objective` is instantiated and the chain of objective components is passed to its constructor. We will show an appropriate source code snippet below.

```
objective = new Objective("obj", Character::interval,
    new ObjSum2Qvel("Qvel", 0., Dof::type_herds[Dof::KINEM]),
    new ObjSum2Qacc("Qacc", 0.00001, Dof::type_herds[Dof::KINEM]),
    new ObjPortDistance("MassCenter", Node::UpdCOM, 1., NULL),
    new ObjPortDistance("ArmCOM", Node::UpdCOM, 1., NULL),
    new ObjDesireDofs("Butt", 1., Character::interval, NULL),
    new ObjPortDistance("ButtOrient", Node::UpdNone, 1., NULL),
    new ObjPortDistance("ShldrTurn", Node::UpdNone, 1., NULL),
    new ObjPortDistance("GroundDist", Node::UpdNone, 1., NULL),
    NULL);
```

This example reflects how the downmapping objective function is usually composed. The names like “MassCenter” and “ButtOrient” explain the semantics of the objective component. The class names are descriptive as well, with names like `ObjSum2QAcc` denoting the sum of the squares of the DOF accelerations (variable Q in Popović’s equations always stands for a DOF).

Figure 3.8 demonstrates a part of the class inheritance tree rooted in class `Func`. These functions are used mainly when setting constraints.

There are a lot of general-purpose functions available. Several of them are specialized subclasses like `Port`, `MassCenter` and `Objective`. The constraint composition with those functions includes three stages. First, the proper constraint activity interval is created. It tells system for which frames the constraint is *active*, i.e. where the constraint must

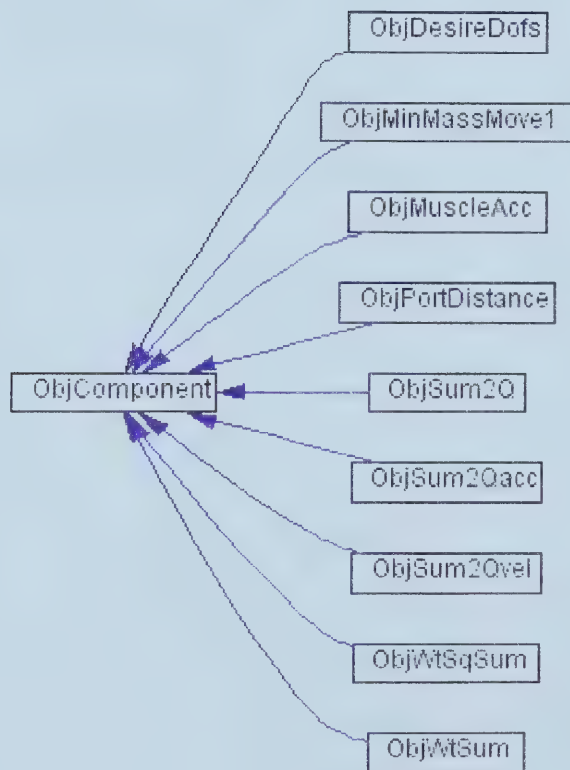


Figure 3.7: Inheritance diagram of the C++ classes representing objective function components

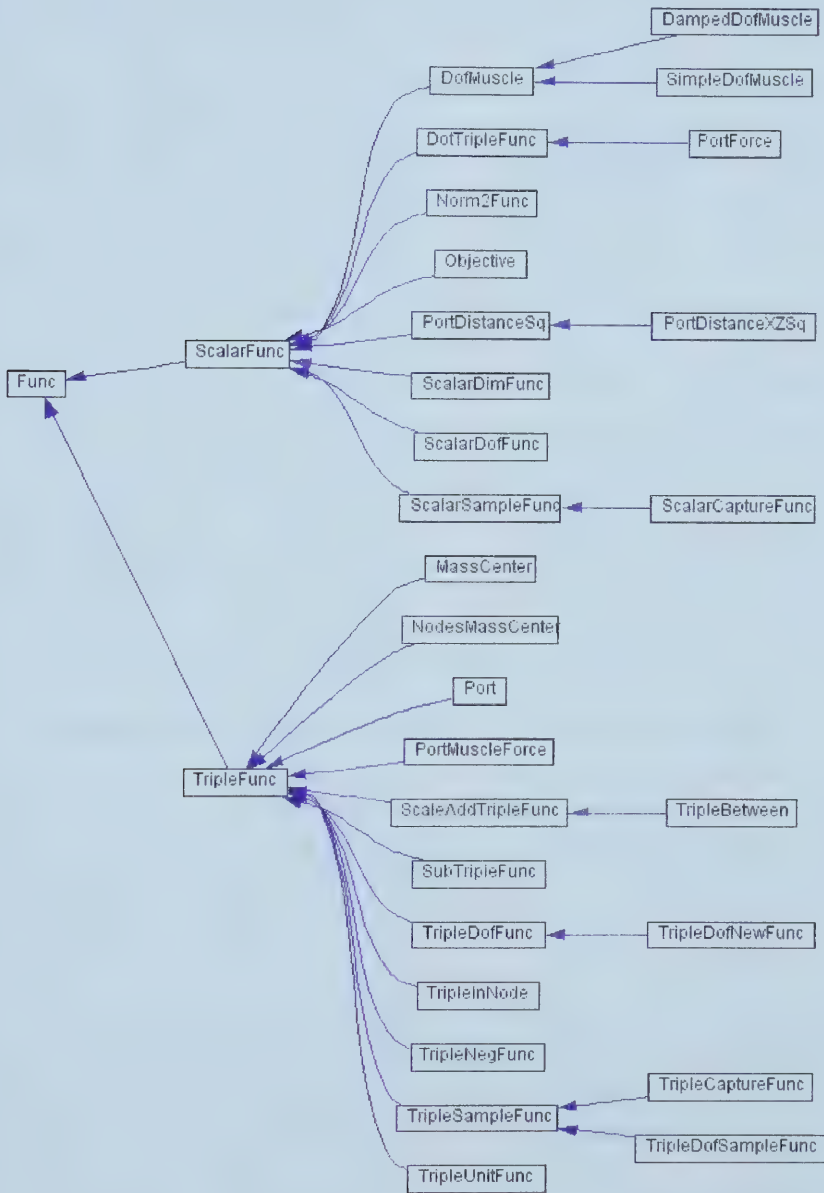


Figure 3.8: Inheritance diagram of the C++ classes representing general-purpose functions available to the user

be observed. Second, the constraint function object must be instantiated and initialized. Finally, the newly created constraint function object must be added to the character's constraint array. Here is the example code snippet.

```
//left foot contact place and time interval
double left_down_start = Character::interval->getClosestSample(0.32);
double left_down_end = Character::interval->getClosestSample(0.48);
Interval *left_down_interval = new RangeInterval("ldown",
        left_down_start, left_down_end-left_down_start, dt);
//create the constraint object
TripleDofNewFunc *lfoot_place = new TripleDofNewFunc(
        "lfoot", left_down_interval, 0.27, 0.02, 0.983);
//add it to the character's pose constraint array
character->addPoseConstr(new ConstrPort(follow_lfoot, lfoot_place, 1));
```

The code above creates the pose constraint necessary for the motion upmapping and dynamic fitting stages. The constraint demonstrated here “nails” the left character's foot to the ground for a short period of time, called a *stance phase* in the biomechanics literature. Without such a constraint, the character's feet would slide on the ground, creating the feeling that the motion is not natural. The pose constraints are not the only type of constraints supported by the MTS. The motion transformation stage involves forces and requires totally different constraint semantics. These constraints are called *mechanical constraints* and are represented by class `MechConstr`.

3.2.3.2 Mapping from the full character onto the simplified one

Let us start with the first phase of the motion transformation process - the downmapping phase. All three motion transformation stages involve certain changes to the source code, loading the appropriate motion data file, and optional problem parameters adjustment through the user interface. Selecting the correct problem type (sequential versus spacetime) and tuning weights of the objective function components are the most often performed operations with the user interface.

3.2.3.2.1 What is exposed to the user Changing the source code to set up the system for the correct motion transformation stage is the most complex operation required on the user's side. Usually, setting up a proper set of objective function components and constraints requires extensive code editing. Sometimes, even the creation of new functions may be required. When mapping motion down to the simplified model, the following four places in code must be changed.

1. The symbol `MAP_DOWN` must be defined in module `main.cc`. The appropriate `#define` directive is already there. The user just has to make sure it is not commented out.
2. The module `io.cc` is responsible for the correct loading of motion data files and *direct DOF substitution* upmapping functionality (see 4.1.2.2 for details). The DOF loading logic is organized so that certain `#define`'s must be commented out for the

downmapping to work correctly. Hence, the user has to make sure all the `#define`'s containing words `UPMAPPING` or `DDS` are commented out.

3. `human.cc` contains the crucial logic of creating a *puppet model* (see the paragraph below about problem setup details and implementation). The user has to `#define MAP_DOWN` there or make sure it is not commented out.
4. The simplified model module (for example, `biped.cc`) has the function creating the *actor model*. The actor model is the model we are going to map motion down to. The symbol `MAP_DOWN` must be `#defined` there as well.

All the above mentioned changes are not too complex to perform when the user can navigate in the code in general. This, of course, requires a fair amount of practice. When the above-mentioned defines are completed, the C preprocessor enables certain sections of code and disables others. As a result, in a compiled form, the system is ready to perform one of the three motion transformation stages. With the four source code modifications described above the MTS is enabled to perform motion downmapping.

After the compilation, the user has to launch the system with the correct set of command line options. We will demonstrate an example for downmapping a jogging motion sequence loaded from the EA mocap studio supplied file. We will not load the whole file. We will load a fragment of the motion, from frame 14 to frame 57.

```
tdmap BPJOG.ea3 14 57
```

`tdmap` is the executable file name, `BPJOG.ea3` is the file containing motion data in EA3 format and the two numbers are the frame range to be loaded. Once the system is started and the motion data file is loaded the user can see the 3D character model in the animation viewer. Now the user can start the downmapping process by pressing the **Solve** button in the main window. Before doing so, the objective component weights can be tuned, using the sliders in the objective control pane in the main window.

3.2.3.2.2 Problem setup details and implementation From the mathematical point of view all the motion transformation stages are nonlinear programs, but the way the nonlinear program is formulated differs significantly from stage to stage. A detailed description of all the three NLP formulations can be found in the Ph.D. thesis of Popović [78]. We will provide a very short, introductory treatment of this issue.

The optimization problem in the form suitable for performing mapping motion down is called a *sequential* problem in the MTS terminology. The sequential problem is a per-frame problem which minimizes difference between pose of the full character and the simplified character. If we minimize the pose difference between two characters for each frame, we make one character mimic the motion of the other one. The reference points used to calculate difference between two poses are called *handles*. For example, we can define an arm mass center for both characters to be a handle. If we have a set of all the handles for the original motion $h_o(t_i)$ and the set of handles for the simplified model motion $h_s(t_i)$, we can define a per-frame pose difference minimization problem as

$$\min_{q_s(t_i)} (h_o(t_i) - h_s(t_i))^2$$

where i is a frame number and $q_s(t_i)$ is a set of all the character's DOFs.

The per-frame problem is handled by the class `ProblemSeq`, defined in modules `problem.cc` and `problem.h`. This class is a mediator between SNOPT and the user-defined model and problem. When the user edits the source code, as described in section 3.2.3.2.1, he/she sets up the optimization problem and constraints to be submitted to SNOPT. `ProblemSeq` is the class that manages calls to SNOPT routines so that the problem is solved for each animation frame.

In Popović's Ph.D. thesis the functions denoted by letter h in the motion mapping problem formulation are called *handles*. In his motion transformation system he uses another term - *guides*. Guides are the vector functions defined on DOFs. An arm's center of mass or a character's center of mass can be called a guide or a handle. Throughout the system's source code the term *guide* can be often met, but the term *handle* appears only in Popović's Ph.D. thesis.

3.2.3.3 Interface with SNOPT

The details of the interaction between SNOPT and the rest of the system (the appropriate C++ classes from `problem.cc` and `solver.cc`) is exactly the same for all the motion mapping, fitting and transformation stages. The problem is represented by the object of class `ProblemSeq` or `ProblemSpacetime`, depending on whether we are dealing with the motion transformation, fitting or mapping stages. `ProblemSeq` is a class designed to represent sequential (in other words, per-frame) optimization problems, while `ProblemSpacetime` serves the same purpose for the spacetime optimization problems.

All the objects of the class inherited from `Problem` contain the member variable `solver` which contains the currently employed numerical solver interface object. For SNOPT this will be an object of class `SolverSnopt`. The second mandatory attribute of all the `Problem` classes is the method `solve()`. This method contains the core logic of the interaction with SNOPT. By looking at its source one can easily understand how exactly SNOPT is set up and what are the semantics of a class representing a particular type of problem. Here is a fragment of C++ code demonstrating how the `ProblemSeq::solve()` method works.

```
INTERVAL_LOOP(Problem::range, t) {

    . . . . .

    //create solver - instance of SolverSnopt in our case
    solver = Solver::create(constr_total,
        coef_total, //number of constraints, number of variable dofs
        constr_total,
        coef_total, //number of nonlinear constraints and coefs is
        //the same, so everything is nonlinear
        constr_types, //constr_eqns in create prototype
        constr_total, //dcdc_sparsity in create prototype
        Objective::obj_active, //has_objective in create prototype
```



```
ProblemSeq::update,    //problem update method (update_f)
this);                //pointer to this ProblemSeq
    //instance (update_d)
. . . . .
//render the model before its pose is "optimized"
anim->glRender(t, 1);
//call the solver (politely)
ret = solver->solve(coef_vals, lower_bounds, upper_bounds);
//render the model again to show impatient user we are working HARD
anim->glRender(t, 1);

. . . . .

} INTERVAL_LOOP_END;
```

The `INTERVAL_LOOP` here is what makes the problem sequential. This loop goes over the animation time interval and solves the pose optimization problem for every frame. The `Solver::create()` method replaces the call to the solver's constructor, automatically instantiating the appropriate solver object (instance of `SolverSnopt` in our case). The solver in its turn has the logic defining how the problem will be converted to the format accepted by the appropriate NLP package. This logic is inside the method `solve()`, a call to which one can notice at the end of the code example above. Two additional `glRender()` calls make sure the animation viewer reflects the process of motion downmapping. The user can see how the simplified model (the actor) mimics the motion of the full model (called *the puppet* in our terminology) as the downmapping process progresses.

The SNOPT interface object (class `SolverSnopt`) with its main method, `solve()` handles all the technical details of the interaction with SNOPT. We will demonstrate a fragment of source code from the method `SolverSnopt::solve()` below.

```
Solver::Return
SolverSnopt::solve(double *x, double *lo_bounds, double *hi_bounds)
{
. . . . .

    //load the SPEC file
    assert(access("snopt.spc", R_OK) == 0);
    unlink("fort.4");
    link("snopt.spc", "fort.4");
    snspeg_(&ispec, &inform, cw, &lencw, iw, &lениw, rw, &lениrw);
    assert(inform == 0);

. . . . .

    //get the SNOPT call parameters ready to fight
    char *startup = "Cold"; //cold startup
```



```

int m = constr_count; //constr_count is constraint
                        //count or 1 if there are no constraints
int n = coef_total;
int ne = dConstr_dCoef->minosSparseCount();
int nName = 1;
int nnCon = constr_total;
int nnObj = coef_total; //all variables in objective are nonlinear
int nnJac = coef_total; //all variables in constraints are nonlinear
int iObj = 0;
double ObjAdd = 0;
char problem_name[8] = "Problem";
char names[] = " ";
int mincw, miniw, minrw;
int nS;
int nInf;
double sInf, Obj;

//the problem is 100% ready, call the doctor
snopt_(startup, &m, &n, &ne, &nName, &nnCon, &nnObj, &nnJac,
&iObj, &ObjAdd, problem_name,
SolverSnopt::snoptJac, SolverSnopt::snoptObj,
a, ha, ka, bl, bu, names, hs, xs, pi, rc,
&inform, &mincw, &miniw, &minrw, &nS, &nInf, &sInf, &Obj,
cw, &lencw, iw, &lениw, rw, &lениrw,
cw, &lencw, iw, &lениw, rw, &lениrw,
strlen(startup));

. . . . .

```

Among many ancillary functions performed by the method `SolverSnopt::solve()` are: the loading of the SNOPT SPEC file (see the SNOPT user's guide [39] for details), final formatting and arranging for the SNOPT call parameters and calling the external the external function `snopt_()` which is the SNOPT's entry point. What happens after that point inside SNOPT is far beyond the scope of this thesis. The final result of the `snopt_()` call is the solved problem, with all the model DOFs satisfying all the constraints and the objective function having the smallest value SNOPT was able to find.

3.2.3.4 Spacetime motion transformation proper

After the model motion was mapped from the full model domain to the simplified model domain, the user can switch to the second stage of motion transformation. The second stage is transforming the motion of the simplified model. Again, as in the case with both motion mapping stages, the user is required to edit the source code, load proper set of motion data files and optionally adjust the problem properties via the user interface. We will get down to it in the following two sections, the third one covers the details of the

interaction between SNOPT and the `ProblemSpacetime` class, the spacetime problem handling C++ class.

Note that before editing constraints and making other changes in a source aimed at making a character to alter the way it moves, the user must perform a preliminary operation called *motion fitting*. This stage is identical to the motion transformation itself, except for constraints. For the motion fitting stage, all the constraints must be set so that the original motion satisfies them. When such a problem is solved, the resulting character DOFs change very little but the motion becomes physically realistic. See Popović's Ph.D. thesis [78] for details about spacetime motion fitting.

3.2.3.4.1 What is exposed to the user The three places in the source code must be edited before the system can be compiled for performing motion transformation (and fitting, since this is essentially a transformation with original, unchanged constraints).

1. The symbol `MAP_DOWN` must **not** be defined in the module `main.cc`. Otherwise, the system will load two 3D models, an actor and a puppet, which will not work for the purpose of motion transformation.
2. The symbol `MAP_DOWN` must be undefined and the symbol `STIME` must be defined in the module containing the simplified model definition that was employed by the user. For instance, if the user chose the `Biped` model, he/she should edit these two `#defines` in the module `biped.cc`.
3. The most important part of the whole motion transformation problem setup is editing the constraints and/or other character related settings. The way constraints are set defines what the final motion will look like. Constraints are the “control knobs” of the motion editing operation. They are not the only knobs that can be employed by the user, but they tend to be the most often used ones. We will give an example of setting a mechanical constraint below.

```
//define mechanical constraint interval
double right_down_start=Character::interval->getClosestSample(0);
double right_down_end=Character::interval->getClosestSample(0.14);
Interval *right_down_interval1 = new RangeInterval("rdown1",
    right_down_start, right_down_end-right_down_start, dt);
//define constraint itself
TripleDofNewFunc *rfoot_place1 = new TripleDofNewFunc(
    "", right_down_interval1, 0.0984, 0.02, -0.435);
//add it to the character's mechanical constraint array
character->addMechConstr(
    rcont = new MechConstr(new
        ConstrPort(follow_rfoot, rfoot_place1, 1),
        Character::interval));
```

Setting the mechanical constraint is very close to that of a pose constraint, the only difference is that class `MechConstr` takes the place of class `ConstrPort`.

3.2.3.4.2 Problem setup details and implementation Unlike the sequential problem used for the purpose of motion mapping, the spacetime problem, represented by the class `ProblemSpacetime`, is not solved for each frame. It is parameterized by time:

$$\begin{aligned} & \min_{q(t)} E(q(t), t) \\ & \text{subject to } \begin{cases} C_p(q(t), t) = 0 \\ C_m(q(t), t) = 0 \\ C_d(q(t), t) = 0 \end{cases} \end{aligned}$$

Hence, it is solved once for the whole animation sequence. $q(t)$ denote character DOFs, C_p, C_m and C_d denote pose, mechanical and dynamic constraints.

Although the nature of the spacetime problem is different from the per-frame problem of motion mapping, the type of solver employed is the same. All the problems in the system, both for motion mapping and transformation, are solved with SNOPT, a commercial nonlinear program solver. See appendix B for SNOPT related information.

The method `ProblemSpacetime::solve()` is short compared with its peer from class `ProblemSeq` because of the relatively simple logic of the spacetime problem setup. This method contains only one important method call:

```
Solver::Return ProblemSpacetime::solve()
{
    double* coef_vals = new double[solver->coef_total];
    double* lower_bounds = new double[solver->coef_total];
    double* upper_bounds = new double[solver->coef_total];

    . . . .
    //important call
    Solver::Return ret = solver->solve(coef_vals,
        lower_bounds,
        upper_bounds);
    . . . .
}
```

As we can see here, the `solve()` method from the spacetime problem class does almost nothing. This comes as no surprise because there is only one problem to solve. The spacetime problem class calls the SNOPT interface class and passes on all the responsibility for solving the problem. We already demonstrated the internals of the SNOPT interface class, `SolverSnopt`, in the section “Interface with SNOPT”.

3.2.3.5 Mapping from the simplified character onto the full one

This motion upmapping phase is the last phase in the process of motion transformation. It alters the original motion of the full character model by applying new DOF data from the simplified model. This is done in the same way mathematically as for the downmapping stage. That is, the per-frame optimization problem is formulated by the user and solved by SNOPT. However, the upmapping problem formulation is quite different from the

downmapping problem. For each animation frame, the following optimization problem is solved:

$$\begin{aligned} & \min_{q_f} E_{dm}(q_o, q_f) \\ & \text{subject to } \begin{cases} C(q) = 0 \\ h_o(q_f) = h_o(q_o) + (h_s(q_t) - h_s(q_s)) \end{cases} \end{aligned}$$

Where:

E_{dm} is the *displaced mass* objective function measuring the closeness between transformed and original motion

q_t, q_s, q_o and q_f are DOFs for transformed, simplified, original and full models.

h_o and h_s are the original and simplified model handles.

$C(q)$ is the set of the full model constraints.

The minimum displaced mass objective solves the task of finding the full model pose given a set of handles. Again, as for the spacetime problem, the solver is SNOPT (see appendix B). It is responsible for finding the full model pose with minimum mass displacement.

The minimum mass displacement objective function allows the system to solve the underdetermined problem, since the number of handles is *considerably smaller* than the number of DOFs in the full model [78]. We have presented a detailed overview of the minimum mass displacement concept in section 1.2.

In essence, the problem of finding the full model motion is almost the replica of the reciprocal problem - mapping motion down. In both cases, we find the optimal model pose for each animation frame. The downmapping process does it directly since the set of simplified model handles determines its pose, but for upmapping process we have a slightly more complex problem to solve. The upmapping handles (also called *remaps* in Popović's parlance) do not define the final model pose directly. They shrink the space of all the possible model poses, while the mass displacement criterion finds the optimal pose in that shrunk space.

3.2.3.5.1 What is exposed to the user For this final stage of the motion transformation process, the user has to edit certain definitions in modules `human.cc`, `io.cc`, `main.cc` and the module containing the simplified model definition. Let us suppose that the user has a `Biped` as a simplified model, then its definition module `biped.cc` would be edited. The list of necessary code modifications is below.

1. As for the motion transformation and fitting stages, the `main.cc` module requires the symbol `MAP_DOWN` to be undefined. It is crucial for proper motion data file loading.
2. The module `io.cc` might require a more extensive editing effort. It has three symbols related to the upmapping stage. The symbol `DIRECT_UPMAPPING` should be defined if the user wants to load the biped DOFs for most major joints (hips, shoulders, etc.) and leave all the other DOFs at zero. This trick will make the full model moving in the same way the simplified model moved. However, since all the

secondary (unimportant) DOFs like the toe DOFs and the wrist DOFs are zero, this motion will not look very realistic.

Another experimental symbol is `HYBRID_UPMAPPING`. Defining it will enable a special mode of upmapping suitable only for the `Walker` model. It loads `Walker` DOFs and combines them with necessary DOFs from the original motion of the full model. The result is that the upper full model part has the original motion, while the bottom part (the butt and everything below it) has the motion of the `Walker`. Thus only the gait of the full character can be changed by `Walker`. See section 4.2.3 for details.

The last symbol is `FULL_DDS`, or “full direct DOF substitution”. The details of this real time method of motion upmapping are given in the section 4.1.2.2. For now, it is sufficient to know that `DDS`, in general, requires the `Biped` model with some skeleton geometry modifications. When `FULL_DDS` is defined, only some secondary DOFs are loaded from the source motion of the full character. Everything else is taken directly from the biped model (hence, the word “substitution”).

Note that these three symbols must not be defined if the user wants to use just the standard `SNOPT`-based method of upmapping, as defined by Popović in [78]. In this case, the user has to undefine all three symbols; for example, by commenting them out.

3. The module `human.cc` requires dealing with two conditional compilation symbols there. One is `HOPPER_UPMAPPING` and the other is `BIPED_UPMAPPING`. One of them must be defined and the other must be undefined. Naturally, the choice depends on the simplified model that was used to transform motion. For the `Hopper` model, the symbol `HOPPER_UPMAPPING` must be defined, while the `Biped` requires symbol `BIPED_UPMAPPING`.
4. Finally, the symbol `MAP_UP` must be defined in the module containing the simplified model definition. Symbols `MAP_DOWN` and `STIME` must be undefined there. Also, the user has to make sure all the transformed motion constraints are identical for both the motion upmapping phase and the spacetime motion transformation phase.

After the system is compiled and ready, the user has to load the file containing the simplified model motion, set the sequential problem type in the problem type chooser located in the main window and press the **Solve** button. After some waiting the user will see the final, transformed motion of the full model. When the motion upmapping phase is over, the user can open up a 2D graph window and use the **File** menu to save the final motion in a file.

3.2.3.5.2 Problem setup details and implementation The C++ code dealing with motion mapping phases is the same. The classes `ProblemSeq` and `SolverSnopt` are used for both downmapping an upmapping. The only significant difference between them is, of course, the objective function. If we look at how the objective function is set up for the upmapping phase, we will notice an additional objective component:


```
objective = new Objective("obj", Character::interval,
    new ObjSum2Qvel("Qvel", 0., Dof::type_herds[Dof::KINEM]),
    new ObjSum2Qacc("Qacc", 0., Dof::type_herds[Dof::KINEM]),

    //butt here means we mimic original Zoran's objective component -
    //full character mass displacement
    new ObjMinMassMove1(character, .2, butt),

    . . . .
```

which is represented by class `ObjMinMassMove1`. The class name is self-explanatory - it is the minimal mass displacement component which we have discussed above. It solves the problem of finding the full character pose which will allow it to keep all the necessary high-frequency motion components intact. Except for this and constraints, one will not find any difference in the problem setup between the downmapping and upmapping phases.

Chapter 4

Video games and motion transformation

In this chapter, we will present our contribution to the area of motion transformation research. Basing on the material of the two previous chapters we will discuss the application of physically based motion transformation to computer games. We will use this term and the term “video games” interchangeably. In the modern world the boundary between traditional game consoles and personal computers becomes more and more blurred and computer games follow the same trend. With such upcoming consoles as Microsoft Xbox [31], the traditional world of video games is going to extend its repertoire far beyond traditional action genre.

With increasingly complex and realistic games, the problem of efficient motion synthesis becomes one of utmost importance. Anywhere in the game where the motion of articulated figures is involved, it is necessary to make the characters move in a way that is close to the way we observe motion in nature. Inherent natural smoothness and grace are very hard to achieve and the task of making motion stylish is in the early stage of research. Certain systems like the “style machines” by Brand and Hertzmann [22] and the MTS by Popović [79] are capable of motion decomposition. A “style machine” can apply the style information extracted from one motion to another, and Popović’s system could do something similar with necessary modification. Still, this is far from the Holy Grail. We are unable to generate motion quickly and efficiently, with minimum manual intervention and with the right amount of style. Every single masterpiece of animation produced so far was done with thousands of man-hours of tedious work or with the motion captured from live performers.

Given all that, it seems for the time being it is a good idea to focus on the animation reuse domain. If it is hard to obtain new quality motion from scratch, is it really easier to recycle the already produced one? The answer is yes and no. Certain motion transformation operations are fairly easy to perform with DOF morphing methods. “Motion warping” is a good example of such a technique [104]. If all the motion transformation ever needed by animators or game developers were limited only to the minor trajectory corrections, the “motion warping” and the methods of similar nature would be the universal answer. Unfortunately, it is often necessary to perform rather significant motion correction. Therefore, motion warping fails as the universal answer. It cannot cope with

changes to the kind of motion with a high “energy level” (or low LOD motion, see section 1.4). That is, the kind of motion requiring the performer’s muscles to release a lot of energy in a short period of time. Any acrobatic jump and many other kinds of motion fall under this category.

The physics-based approach has its own problems. If we look at spacetime motion transformation, we will find it slow and cumbersome in setting up the motion transformation problem to solve. On the other hand, in the world of computer games the ultra-high quality of the “spacetime transformed” animation is not the first feature a game designer would want. The notable quality of the action computer games always was its reliance on the deep immersion of the player in the game world. It turned out that people went absolutely crazy even after such a primitive first person shooter as Doom [49]. This game did not even have motion in the sense we understand it now. The characters in the game were not 3D models and there were no DOFs or animation channels, or anything we expect from the modern game with articulated characters in it. The simple 2D sprites with primitive (although, quite appealing and impressive) animation created the worldwide phenomenon. Other famous old-time 3D game classics like Descent [88], also do not have the motion of articulated figures in them but they still can impress many of us by their art, music and the whole game atmosphere.

From the summary above, we can state that high-quality motion is not the primary goal of many game development teams. A good game has a lot of factors determining its success and the motion of the characters is only one of them. In some cases, such as sports simulators, the motion is observed from a distance or there are too many moving figures to focus attention on one of them (soccer immediately comes to mind). In other cases, the characters of the game have a low polygon count and this is compensated by the texture art on their bodies. “American McGee’s Alice” [4] is an example of such a technique. Another, even better example is the 3D action game “Max Payne” [34]. In this game the extreme realism¹ is achieved by mixing smooth and vivid skeletal animation with ultra-high level of texture detail combined with advanced techniques in 3D rendering and lighting. However, the groundbreaking level of realism in this game is not the only feature setting it above other titles of similar nature. The comics-style story of the game and the whole plot-driven action are the real distinguishing features of the “Max Payne”. The relatively high-quality motion plays only so much in bringing this game to its “top of the crop” rank.

Judging by the examples above, one can conclude that, in general, sophisticated, “liquid”, graceful and stylish motion alone and by itself will not give major advantage for a good 3D computer game. Therefore, we can try to simplify things a bit and combine a powerful and high quality spacetime motion transformation with less precise but much faster techniques from the area of motion warping and similar methods. This is what this chapter is about.

We will discuss the possible application of spacetime motion transformation to the domain of computer games. First, we will cover what we can do regarding the task of motion retargeting; then we will discuss the ways we can speed things up. Finally, to prove the viability of mixing spacetime transformation with simpler techniques, we will

¹At the time of writing this thesis “Max Payne” was a landmark of realism in 3D action games.

demonstrate the set of three models from the physically based motion transformation system by Popović, covered in detail in the previous chapter. The demonstration will not be limited to the models only, we will show what we can do about speeding up the third phase of the motion transformation process, the motion *upmapping* phase. We will also present our ideas regarding the model design for the MTS. Real time motion upmapping cannot be done with any model, there are guidelines the model designer should follow to be able to use it. We will show and explain those guidelines in detail.

The presentation of the motion transformation concepts would have been incomplete without the animation itself. Therefore, the author of this thesis provided for a set of animation clips online [96]. Appendix D contains the list of the animation clips online.

4.1 Using physically based motion transformation in video games

The MTS by Popović takes the motion data produced by mocap as its input. This is not a coincidence, even though it does not matter where the input motion came from. The primary goal of the Popović's research was to find a method for making captured motion data editable. By making it editable, he would overcome one of the largest flaws of motion capture - the static, non-editable nature of the data obtained. However, if one looks at the state of motion storage and playback in the modern 3D computer games, one can easily identify the same flaw the games suffer - static motion. Another problem which follows from the static nature of the motion is the tremendous space requirements to store all the motion that the game will need to animate its characters. The problem of motion storage is the consequence of the static nature of motion data in the game. For every minor deviation in motion that is necessary, the animator has to include a whole new animation sequence in the game. It seems that the idea of *motion retargeting*, i.e. adjusting motion of the character to the changes in its environment, could yield both a decrease in the space requirements for the game data and an increase of the realism in the game.

The next two sections focus on these two would-be advantages separately. First, we discuss what could happen if real time motion retargeting would be available to the game designers. Second, we look into the problem of motion data storage requirements. We will see if it is possible to do something about it, even if we do not have any motion retargeting functionality in the game.

4.1.1 Dynamic motion retargeting

We already mentioned problems with the motion transformation system by Popović in chapter 1. Speed is a primary concern for the game development teams. With SNOPT as a plugged-in commercial solver, the MTS can serve only as a research tool. For production purposes, the nonlinear solver must be replaced with the custom solver, taking advantage of the knowledge of problem specifics. For example, we can limit the solver to a subset of the frames in the animation sequence. We could use interpolation to mix the modified frames data with all the frames excluded from the transformation process. In other words,

we could lower the resolution we are working with. Instead of including every frame in the spacetime problem, we can try to include every second frame or even every third frame. Other options include multigrid [21] [23] and multiple shooting [55] methods.² There is also a possible way of speed improvement regarding the “motion resolution”. At this moment the MTS works with resolution uniformly distributed in time. That is, the frame rate of the motion the MTS is transforming is constant over time. One could try to make that attribute variable, lowering resolution for certain fragments of the animation and increasing for others. This approach requires extensive research on his own, but it might be worth it.

Anyway, let us suppose we have a real time nonlinear program solver in our arsenal. What can we do with the motion stored in the game now? First, we can do real time motion retargeting. In terms of the MTS, we can change the objective function, solve the problem and get a motion that has a “tired” look, or a “brisk” look, or any other style we can achieve by varying objective function. Alternatively, we can change constraints and now the style is intact but the character still moves differently. If we change both the objective function and constraints, we will have a character whose motion style and the motion itself have changed. Thus, in situations where the football player has to change gait or reach another point in space to catch a ball we do not have to keep all the imaginable motion clips in the game. The real time motion transformation system will do the job and will quickly produce high-quality realistic motion satisfying all the constraints we need.

The second advantage, which is a direct result of the first one, is the economy of storage space needed for keeping all the animation data of the game. By grouping the motion clips into categories separated by the motion type (jump, run, walk, bend, fall, etc.) and eliminating the ones that are close enough, we can achieve significant economy of space taken by the animation data. Whenever the game character has to perform multiple motion sequences that differ only by the style and/or by the constraints, we can replace the whole set of motions by one plus a set of constraints and/or the style relevant information. Sometimes the constraint information or the necessary motion style is determined dynamically in the course of gameplay. In such a situation, we will need even less space for storing the motion. We can save the source motion sequence and that is it. The real time motion transformation system in the game will produce the final motion data from the source motion and constraints (or style requirements).

Unfortunately, such a nice addition to the palette of game development tools is nothing more than a pleasant dream at the moment. The problem of bringing the NLP solver to real time is very hard and requires effort far beyond the limits of this thesis research. On the other hand, there are other weak spots in the system which we can attack. One of them is the third stage of motion transformation - the upmapping stage. All three stages of motion transformation are formulated as a kind of mathematical program. The validity of such a formulation for the second stage is beyond any doubt. The whole system has been built on the idea of formulating the task of motion transformation as a mathematical program. However, there are no principal requirements for the first and the third stages also to be formulated as mathematical programs.

²Suggested by Popović himself.

Once the lack of these principal requirements is understood, the next logical step is to think about alternative formulations of the first and the third motion transformation stages. Naturally, we have to think of an alternative way of motion mapping which will speed it up. The next section discusses such an alternative way of mapping motion up. Note that the real time motion upmapping idea is also applicable to the downmapping stage. We will discuss that in section 5.3.1.

4.1.2 Real time motion mapping

4.1.2.1 Motion “frequency”

The task of motion upmapping is to blend the two moving 3D characters so that the simplified character becomes the “driving force” of the full character. The result of such a blending operation is a new motion which combines the important (or *low frequency*) characteristics of the simplified model motion with the additional or *high frequency* characteristics of the full model motion.

We have to illustrate the notion of low and high-frequency motion. The terms used to denote them are rather unfortunate, but we will stick to them anyway. Figure 4.1 demonstrates two models used in the MTS. On the left side is the **Hopper** model, on the right side is the **Human** model.

The **Hopper** is a simplified model used to work with jumping motion. It has only four DOFs, except for global translation and rotation. The **Human** is a full model, so it can represent almost any motion. Between those models one can see the list of DOFs. The DOFs on figure 4.1 are typeset in fonts of different sizes. The largest font is set for the global DOFs. These six DOFs, three for translation and three for Euler rotation, are responsible for the motion of the model as a whole. Smaller fonts are used for hip and abdomen DOFs for the **Human**, and for center of mass (COM) and leg DOFs for the **Hopper**. The smallest font is set for the knee and shoulder DOFs in the **Human** model. The font size is proportional to the importance of the DOF in the model hierarchy. The “importance” of the DOF in the hierarchy depends on the scale of *mass displacement* caused by the variations of this DOF. See section 1.2 for the explanation of the mass displacement concept.

Let us start from the root of the hierarchy tree of the **Human** model. The root node is the butt node. Therefore, according to the mass displacement definition, any change in one of the six global translation and rotation DOFs will propagate down the whole tree. That is why the global DOFs are the most “important”. Going up or down from the butt, we encounter either abdomen or hip DOFs. A change in one of these will propagate only to the fragment of the model tree. Hence, the hip and abdomen DOFs are not as “important” as the global DOFs.

Continuing this analogy we will eventually develop a rule of DOF importance. It is quite simple - the closer the DOF is to the root, the more important it is.

Switching back from the mass displacement and DOF importance to the low and high-frequency motion, we can now draw a parallel between DOF importance and motion “frequency”. So-called “low-frequency” motion is just another term for expressing variations of “important” DOFs. The opposite - the “high-frequency” motion, is the

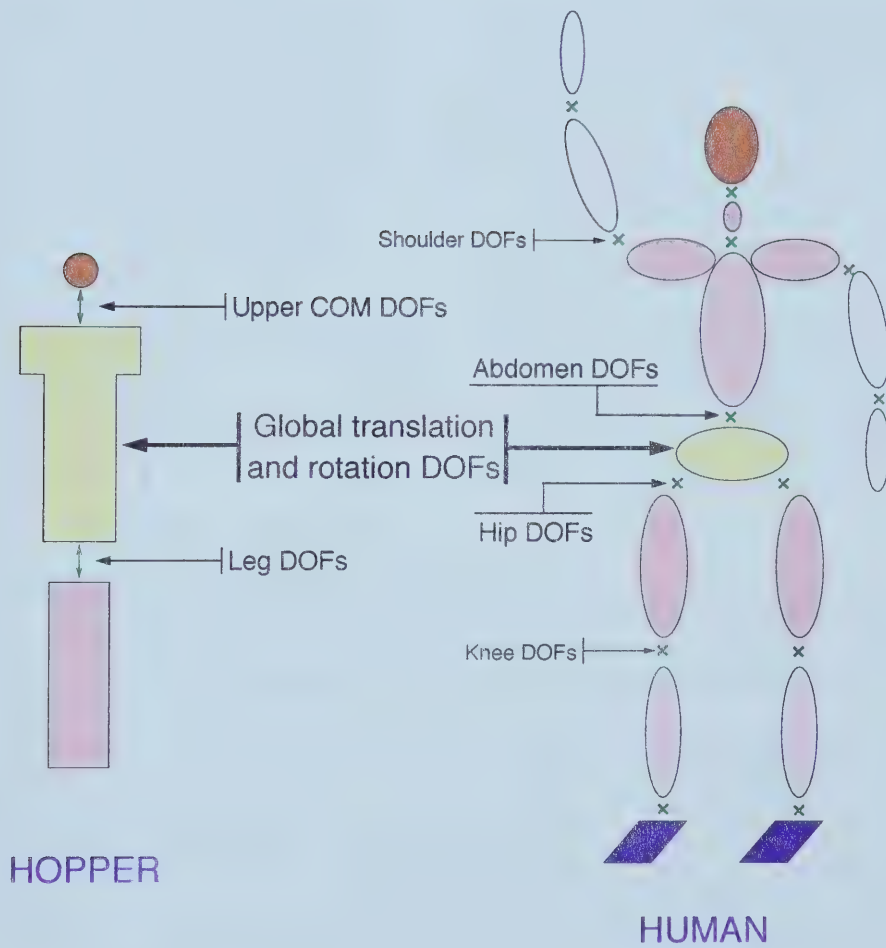


Figure 4.1: Low and high-frequency components of motion

alternative name for changes in less important DOFs, such as the wrist or ankle joints of the **Human** model. Probably, it would be better to replace imprecise classification by imaginary “frequency” with a mass displacement-based classification. “High mass displacement motion” and “low mass displacement motion” terms would make much more sense.

Such a motion classification does not seem to be useful for our purposes unless one interesting feature is noted. If we look at the highly dynamic motion, for example, a jump, it is easy to notice that the low-frequency (or high mass displacement) motion captures the dynamic characteristics of motion, while high-frequency tends to focus on smaller details. It is obvious that motion of the fingers during a jump is a small and irrelevant detail. It is not irrelevant from the artistic point of view, as this finger gesture may convey some important information or expression in the course of animation. It is irrelevant from the point of view of motion dynamics. Whether the finger on the character’s hand moves or not, the way the character jumps will not change if any kind of simulation of motion physics is employed to derive the motion. The mass of the finger is so small compared with mass of other parts of human body that it does not influence motion at all.

So, how can we exploit the fact that the DOFs of the character can be separated into two groups, one essentially defining the motion while the other merely adding some unimportant details? The next section answers that question by presenting a method for the real time motion upmapping. The theoretical foundation of that method is the DOF separation into “important” and “unimportant” groups.

4.1.2.2 Direct DOF substitution

The standard method of motion upmapping employed by the MTS involves solving a mathematical program for each animation frame. The goal of this program is to find a new pose of the full character with minimal mass displacement compared with the original pose, while observing constraints set by the handles of the simplified model. See section 3.2.3.5 for details.

The process of the frame-by-frame motion upmapping creates all the full model DOFs from scratch. In the process, all the low-frequency DOFs are mostly defined by the handle constraints while the high-frequency DOFs mostly follow their original trajectories from the full model. Observing the transformed motion of the full model and comparing it with motion of the simplified model, one can immediately notice this. The legs, torso and arms of the full model almost precisely follow those of the simplified model while the motion of feet, head and palms is generally the same as in the original motion of the full model.

Once this is noted, the direct DOF substitution idea becomes obvious. If the low-frequency DOFs follow their counterparts from the simplified model and the high-frequency DOFs are essentially the replicas of their counterparts from the full model, we can obtain transformed motion by combining two sets of DOFs. In other words, we can *substitute* DOFs from the transformed motion of the simplified model for the corresponding DOFs of the full model. Figure 4.2 illustrates the DDS concept.

On this figure, one can see how DOFs of the two models are combined to produce the transformed motion of the full model. The simplified model is a **Biped** and the full

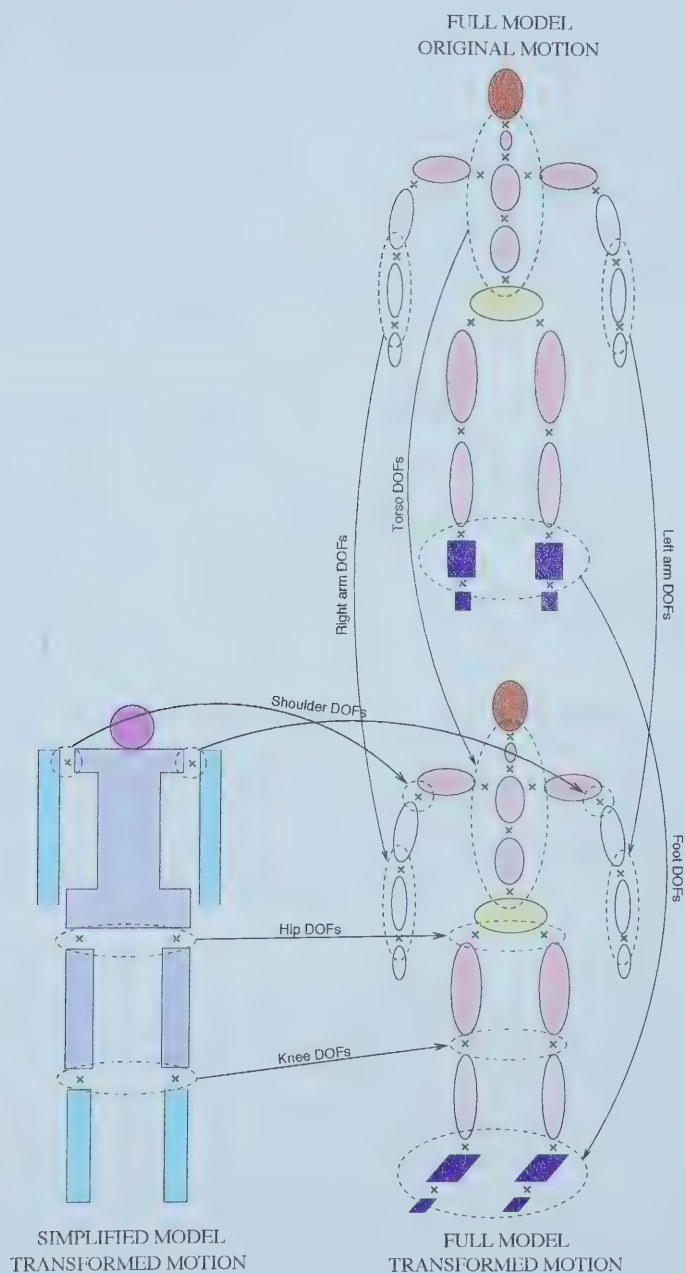


Figure 4.2: Restoring motion from Biped to EAHuman by direct DOF substitution

model is an **EAHuman**. DOFs belonging to **EAHuman** contain the original motion. The DOFs of a **Biped** contain the transformed motion. Substituting the **Biped**'s DOFs for the corresponding DOFs of a **EAHuman** model, we will get **EAHuman** moving exactly as the **Biped** moves; but with all the fine details like wrist or head motion left intact.

The curious reader will probably ask at this moment, "How about other models, like **Hopper**?" Indeed, this is a very interesting question and the next installment of four sections covers the results of experiments with various simplified character models. Four models have been used in the experiments. Two of them have been taken from the set of simplified models supplied by Popović with his motion transformation system. The other two were created by the author of this thesis specifically for the purpose of exploring the limits and capabilities of the direct DOF substitution.

4.2 Experimenting with models

First, we have to note that DDS will not work for every possible combination of the simplified and full model. This is an inherent and unavoidable weakness of DDS. For the substitution of DOFs to work, the simplified and the full models must obey certain rules. There will be more on that in section 4.2.5.

Second, the idea of DOF substitution can be naturally extended to the downmapping phase. However, we do not provide any experimental data on the downmapping process here. Everything that will be said about substituting DOFs between models is true for downmapping too.

We will start our review with the simplest model, called **Hopper**. Next follows the model which is the most suitable for DDS - the **Biped**. The last two models are the creations of the author of this thesis. **Walker** is a simplified model specialized for transforming gaits and **Torso** is designed for the independent transformation of the upper human torso motion. We will also cover the issues of the design of simplified models, from the point of view of making them suitable for the DDS method.

4.2.1 Hopper

This model was designed by Popović as a test of the limits of the model simplification. The **Hopper** is a very simple model. It has only 10 DOFs, 6 of which are global. The price for this simplicity is the extremely narrow range of motion which can be represented by this model. The **Hopper** has only one leg with one degree of freedom. Hence, it can only be used for representing jumping motion. You can see the **Hopper** on figure 4.3.

Apparently, the **Hopper** is not very suitable for DDS, to say the least. For example, the only leg DOF cannot be directly substituted for any DOF in the full human model. It is still possible to obtain knee and hip DOFs, if we put some limits on them. Then we could employ IK to translate the linear hopper leg DOF into the angular DOFs of the human hips and knees.

The problem with the leg DOF is, however, nothing compared with the problem of restoring the motion of the upper torso during upmapping. The human torso has at least 13 DOFs, while **Hopper**'s upper COM (center of mass) has only three DOFs. There

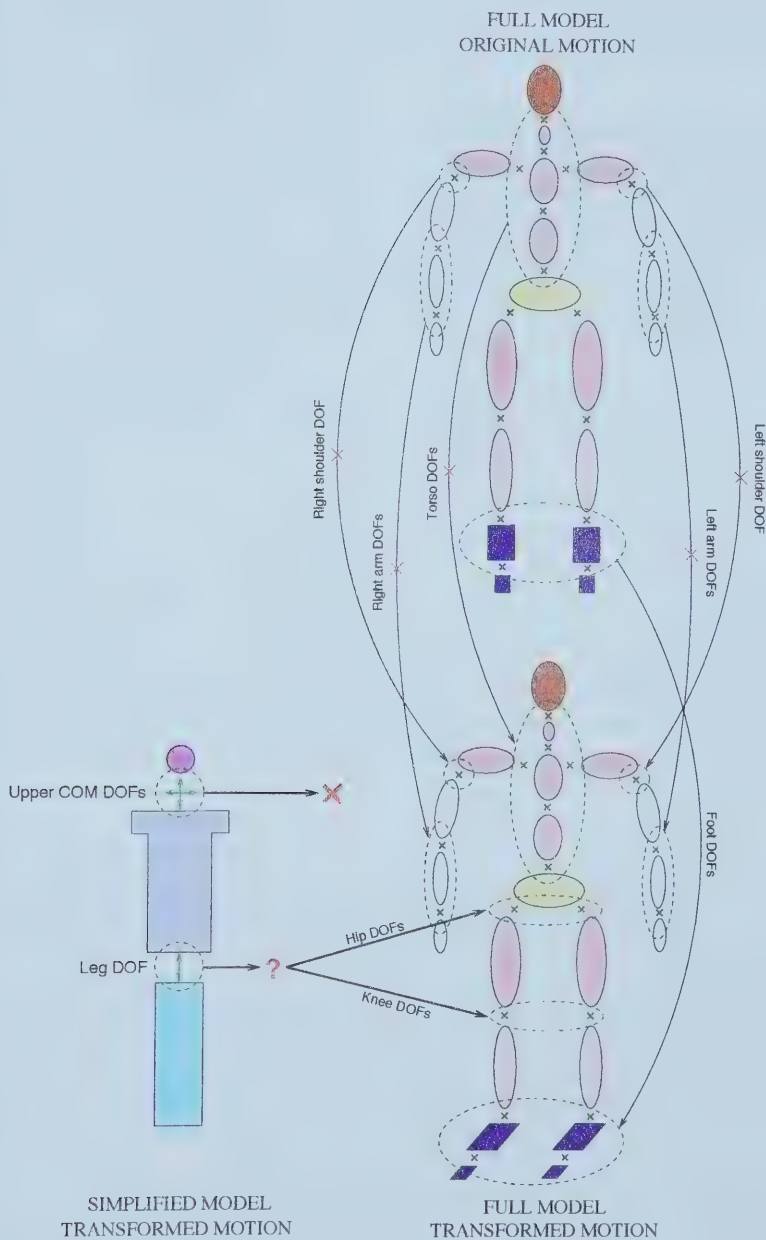


Figure 4.3: Restoring motion from Hopper to Human by direct DOF substitution is impossible

is no way the change in COM DOFs can be unambiguously mapped to the set of 13 human upper torso DOFs without employing the mass displacement criterion. Mass displacement means using the numerical solver again, and the whole point of achieving real time upmapping evaporates instantly.

There are ways to speed up the standard method of upmapping for the **Hopper** model. One is the obvious frontal attack approach - get the speed of the numerical solver down to the acceptable value by writing a custom solver. Another, much simpler one is to play with the SNOPT configuration file and change the number of maximum allowed iterations per SNOPT run. Thus, it is possible to decrease the time SNOPT spends on solving frame optimization subproblems; but it also causes the quality of the solution to be compromised. There is a balance where the quality of the final motion is still high, while the time spent on motion upmapping is as short as possible. Finding such a balance requires some experimentation though.

We will discuss this issue in more detail later in section 5.2.1.

4.2.2 Biped

The **Biped** model comes as the most complex model in the set of simplified models supplied by Popović with his motion transformation system. It is versatile enough to represent any kind of highly dynamic motion. It is able to capture the dynamic essence of any walk, run, jump, vault and other kinds of dynamic motion. The opposite side of such a versatility is the number of DOFs the **Biped** contains. To be able to represent such a wide range of animations, the **Biped** has to have at least 20 DOFs including 6 global ones. This makes the spacetime problem much larger than it is for the **Hopper** model. It can be said there is a tradeoff between the speed of the spacetime motion transformation phase and the speed of the motion upmapping phase. Making the simplified model complex enough to be suitable for DDS causes the spacetime phase to become longer, but the upmapping time essentially drops down to zero.

Figure 4.2 shows how the DDS is set up for the pair of **Biped** and **Human** models. All the DOFs from **Biped** are directly mapped to their peers in the **Human**.

We have applied the DDS upmapping to the “wide running” and “criss-cross running” animations, similar to those from Popović’s Ph.D. thesis but taken from the EA mocap studio. The original normal run animation scene has been transformed to criss-cross and wide run animations using foot-ground contact constraints. First, we produced the transformed motion with the standard method and then we performed direct DOF substitution and compared the results. Just by looking at them, the two transformed animations appear so similar that it is impossible to say which was upmapped by SNOPT and which by DDS. The differences are negligible.

Below is a series of animation frames demonstrating the motion produced by the standard upmapping and by the DDS upmapping. It is impossible to notice a difference on pictures and it is almost as hard for the animation clips. Figure 4.4 is the “wide run” animation upmapped by the standard per-frame problem solving. Figure 4.5 is the same motion but now it is upmapped by the direct DOF substitution.

Figure 4.4 corresponds to clip 7 in table D.1. Figure 4.5 corresponds to clip 11 in table D.1.



Figure 4.4: EAHuman wide run upmapped by the standard per-frame problem solving

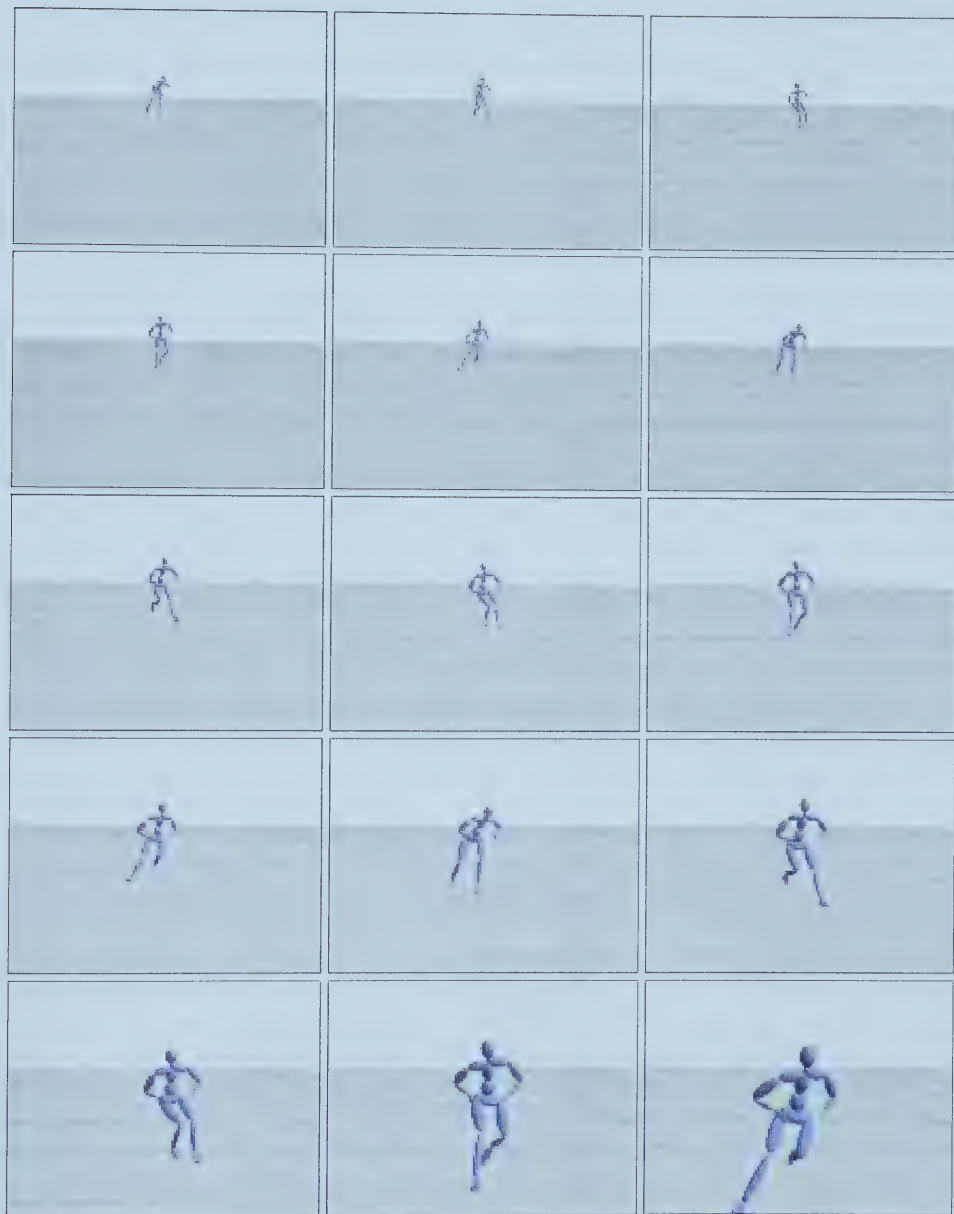


Figure 4.5: EAHuman wide run upmapped by the direct DOF substitution

What cannot be observed on the pictures is the foot sliding appearing in the final motion when the DDS upmapping is used. This sliding looks as if the character runs on the ice and does a poor job trying to keep its balance. It is a serious flaw and we had to devise another simplified model to fix it. The next section elaborates on that.

4.2.3 Walker

The **Walker** model is a pair of legs, a butt and a COM above it. The COM represents the whole upper torso, just as it is done in the **Hopper**. One can think of a **Walker** as a mix between a **Biped** and a **Hopper** but there is the other important feature besides the COM representing the upper torso. The feet of the **Walker** are the precise replicas of the feet of the **Human** model. Naturally, since the level of detail of motion representation has increased for feet (**Biped** did not have toes and associated with them DOFs), the DDS mapping of DOFs has changed as well. Look at figure 4.6.

Here, one can notice two interesting details. First, the legs of the **Human** model and the **Walker** model are exactly the same. Therefore, with DDS mapping of DOFs we will achieve the absolutely correct motion of the feet, while observing all the foot-ground placement constraints. This fixed the problem with sliding feet in the motion DDS-upmapped from the **Biped** (see section 4.2.2). Second, there is no DDS mapping information regarding upper COM DOFs. The reason for this is that the DDS is happening only for the leg DOFs; the upper COM DOFs are ignored. The upper torso DOFs, taken from the original motion, are used instead. That is, the motion of the upper torso has not changed at all.

This should come as a bit of surprise. The reader might have noticed the similarity between the **Hopper** and the **Walker** in the upper torso representation and is already questioning the very different treatment of the upper COM DOFs mapping for both cases. Indeed, why do we ignore the upper COM DOFs for the **Walker** and cannot do the same for the **Hopper**?

The answer lies in the nature of the jumping and the running motion as well as the role the upper COM plays in both of them. For the jump, the COM DOFs are significantly changed whenever the jump itself is significantly changed. For the run, the picture is the opposite - the upper COM DOFs stay almost constant when the foot-ground placement constraints are adjusted. This is, of course, not always true. There are cases when editing a jump would not make the upper COM DOFs vary noticeably. On the other hand, the editing of a run can be done so that the upper COM DOFs change quite a bit. Still, the general “rule of the upper COM DOFs” for the jumps and runs/walks is true.

The experimental results prove it too. Look at the series of frames from the two animations on figures 4.7 and 4.8. The former is the **Walker**’s criss-cross run upmapped by the standard SNOPT-based per-frame problem solving method. The latter is the same motion but upmapped with DDS. They look the same on the pictures and they certainly do in the animation itself.

Figure 4.7 corresponds to clip 13 in table D.1. Figure 4.8 corresponds to clip 26 in table D.1.

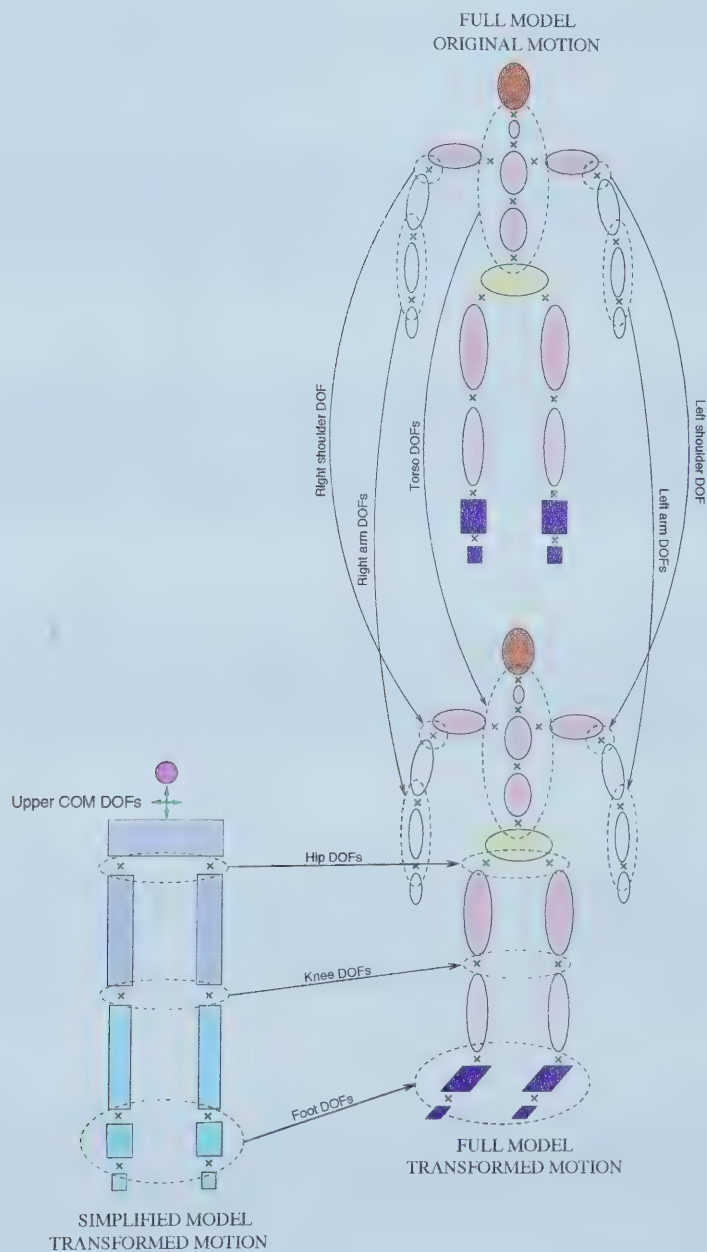


Figure 4.6: Restoring motion from Walker to Human by direct DOF substitution



Figure 4.7: Walker criss-cross run unmapped by the per-frame problem solving

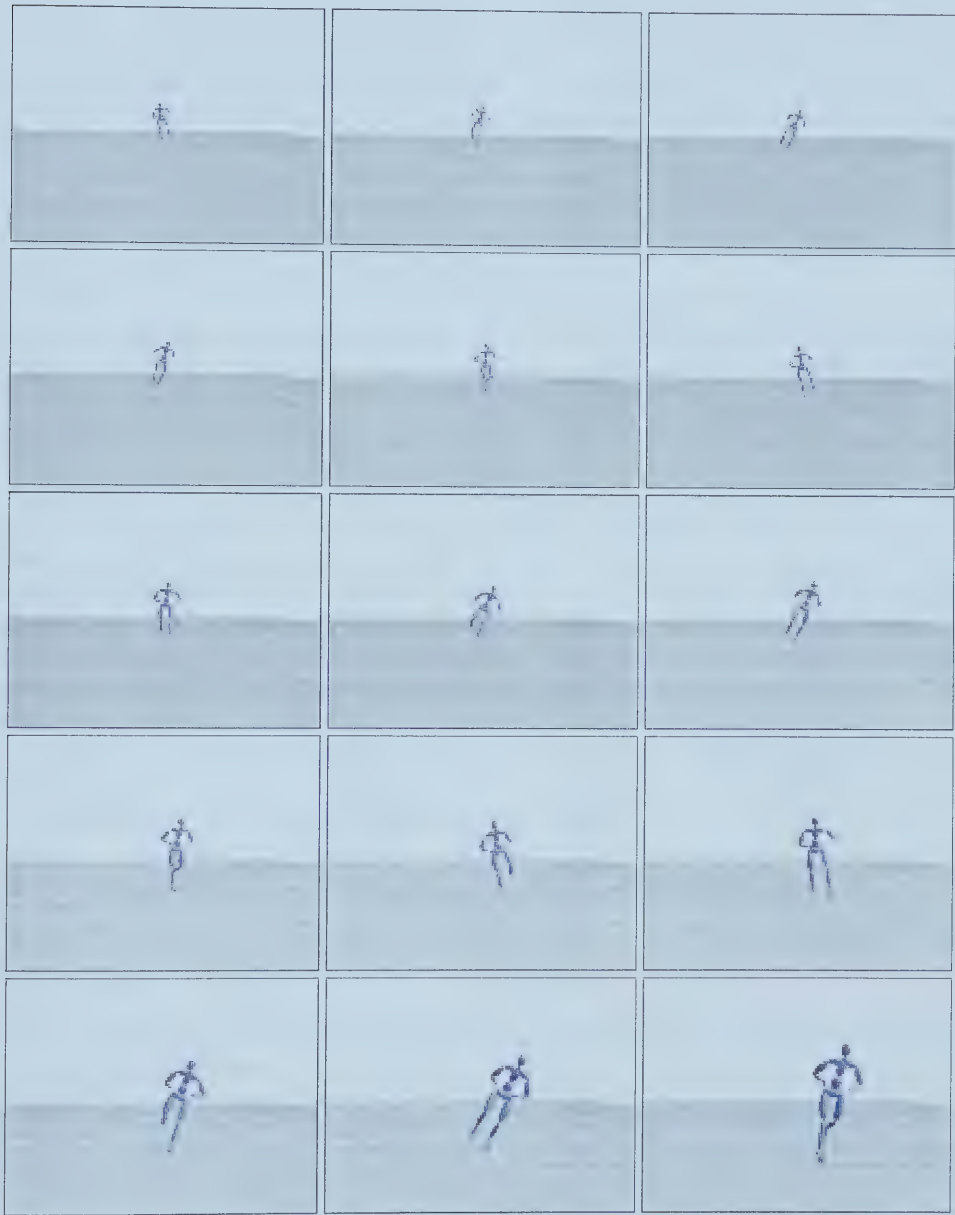


Figure 4.8: Walker criss-cross run upmapped by the direct DOF substitution

4.2.4 Torso

This is the second model that has been added to the palette of the simplified model provided by Popović with his system. Figures 4.9 and 4.10 show the DDS upmapping for Torso and 3D image of this model. Clearly this model is designed for the task diametrically opposite to that of the Walker. It can be used to change the motion of the upper torso only.

As with all the previous models this one also has a special feature. The Torso is the only model that represents only a *fragment* of the full skeleton. There is no such thing as a “lower COM” or something like that. The Torso model is just what its name says. It is an upper torso of a human body, with no additions.

“What is it useful for?”, the reader is probably asking now. The answer is easy to guess - the Torso can be employed anytime the animator is concerned about changing the motion of any part of the human upper torso and is not interested in editing DOFs below the butt. Figure 4.9 demonstrates the DDS mapping used for this model. Everything is obvious - all the Torso DOFs are directly mapped to their peers from the Human model.

Here is the series of frames from the original normal run animation (figure 4.11), the two downmapped motion sequences - for the Torso (figure 4.12) and for the Walker (figure 4.13), the edited motion of the Torso (figure 4.14) and, finally, the combined motion of the Walker and the Torso models (figure 4.15). As it is obvious from the frames, the idea of slashing the full human body model into Walker and Torso works just fine. The animator can construct new motion by combining the motion of these two *submodels*. There is, of course, the issue of the motion synchronization between the two independent submodels, but it is not fatal. Synchronization can be added here, but it is beyond the scope of this thesis.

Table D.1 contains descriptions for all the five animation clips corresponding to the figures above. Figure 4.11 corresponds to clip 2, figure 4.12 - to clip 32, figure 4.13 - to clip 35, figure 4.14 - to clip 33 and figure 4.15 - to clip 28.

4.2.5 Guidelines for designing models suitable for DDS

It is time to summarize the results of the experiments with the four simplified models described above. We can draw the following conclusions quite definitely:

- The DDS upmapping is a real time motion upmapping technique, ready to be used in computer games.
- The DDS upmapping has a nice side effect - it can help to conserve the space taken by the animation data inside the game. This is actually a good topic for a whole new research project, but even now it is obvious that certain kinds of motion can be stored as a set of full model motion plus several so-called “deltas”, which are the simplified model motion sequences. The full model motion can be restored in real time from the original full model motion plus the necessary delta. Also, one can store a set of submodels and construct the new animation sequences on the fly, just like it has been done for the Walker and Torso pair.

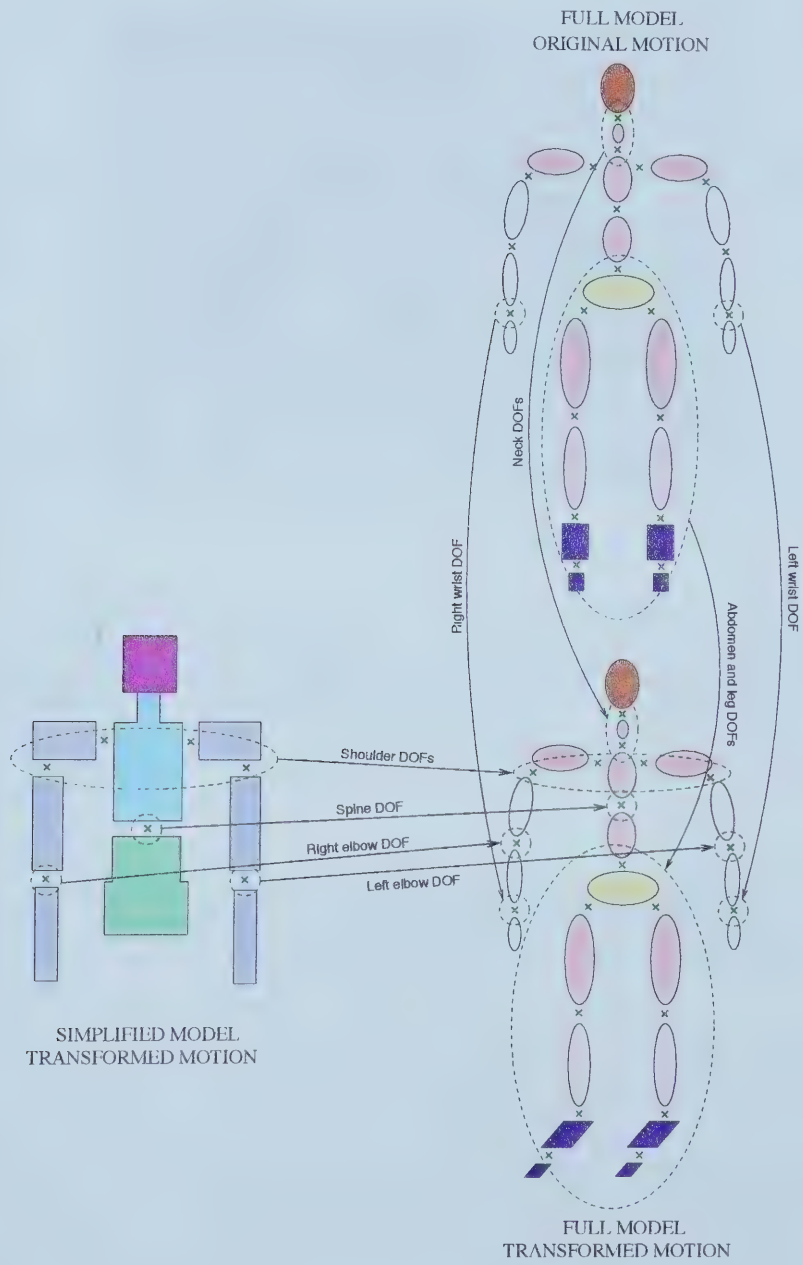


Figure 4.9: Restoring motion from Torso to Human by direct DOF substitution



Figure 4.10: 3D image of the Torso model

- The DDS upmapping has a drawback - it will not work for any pair of the simplified/full models. DDS requires a pair of models specifically tuned for it. There are certain guidelines the model designer must follow to make it possible to upmap motion in real time using the DDS method. These guidelines are presented below.

Figure 4.16 visualizes the set of rules to be followed when designing the pair of simplified and full models.

There are three rules to follow.

1. The model geometry must be exactly the same for all the nodes where the DOF substitution to happen. We are talking about the bone length here. The width can vary, and since it does not influence the motion of the bone ends, it is just a matter of choice to make the model visually appealing.
2. The translation at the joints has to use the same units and rotation there has to be of the same type. In other words, if one model uses Euler rotation at some joint, the other model has to use the Euler rotation too, not the quaternion form.
3. The order of translation and rotation has to be exactly the same for both models. Of course, this rule applies only to the places where the DOF substitution happens. Note that in some cases it will be impossible to substitute DOFs only for the limited set of joints. Depending on how the DOFs are represented for both models it may be necessary to substitute all the DOFs down the hierarchy from the place where such a substitution is really needed. A good example of this was the experiment with the Walker model, where the Walker's DOFs were of the type different from those



Figure 4.11: EAHuman normal run, as provided by EA mocap studio



Figure 4.12: Torso motion taken from the normal run of EAHuman



Figure 4.13: Walker normal run downmapped from the EAHuman motion



Figure 4.14: Torso's motion is now transformed - it is waving its hands

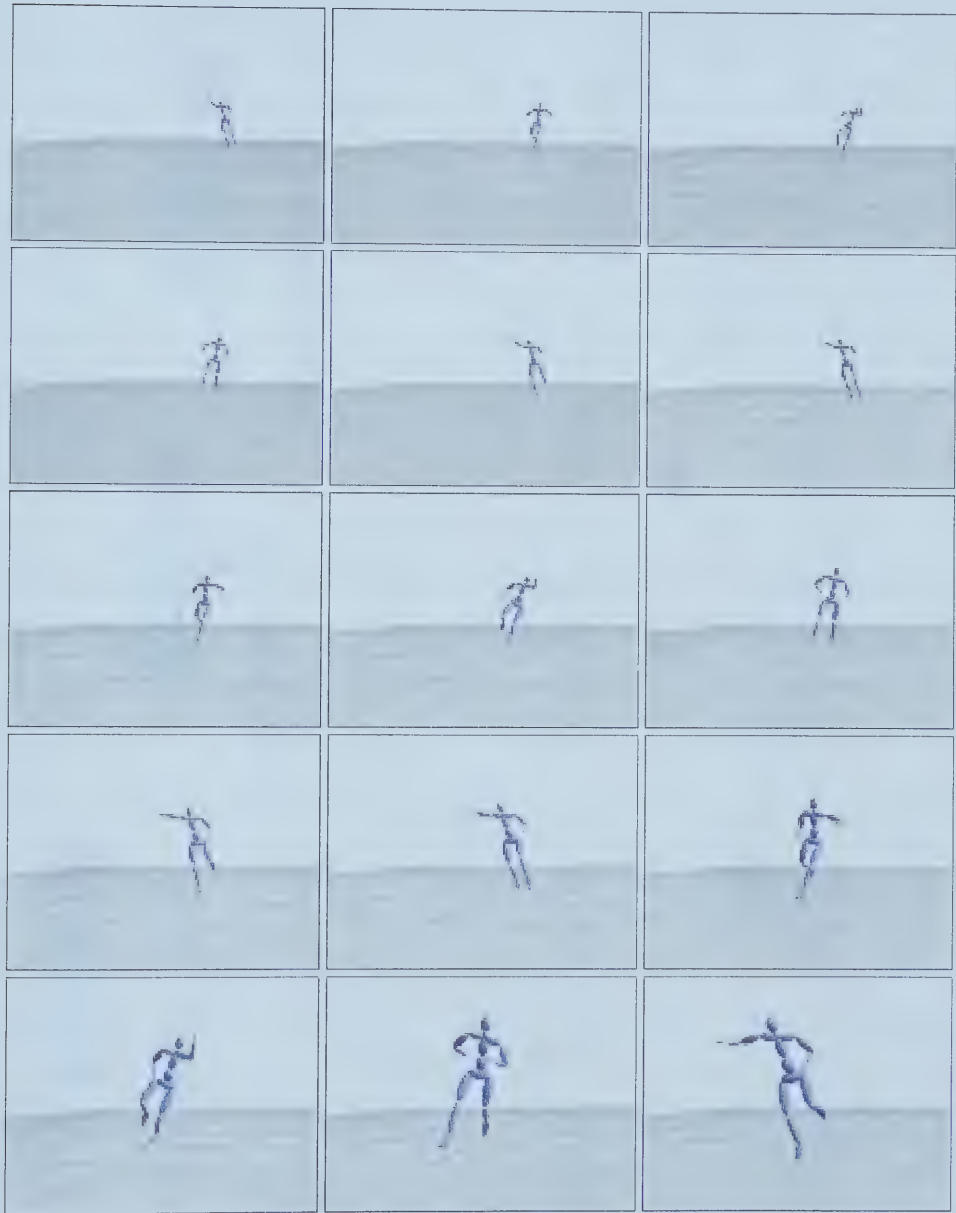


Figure 4.15: Walker criss-cross run and Torso “salute” motion combined

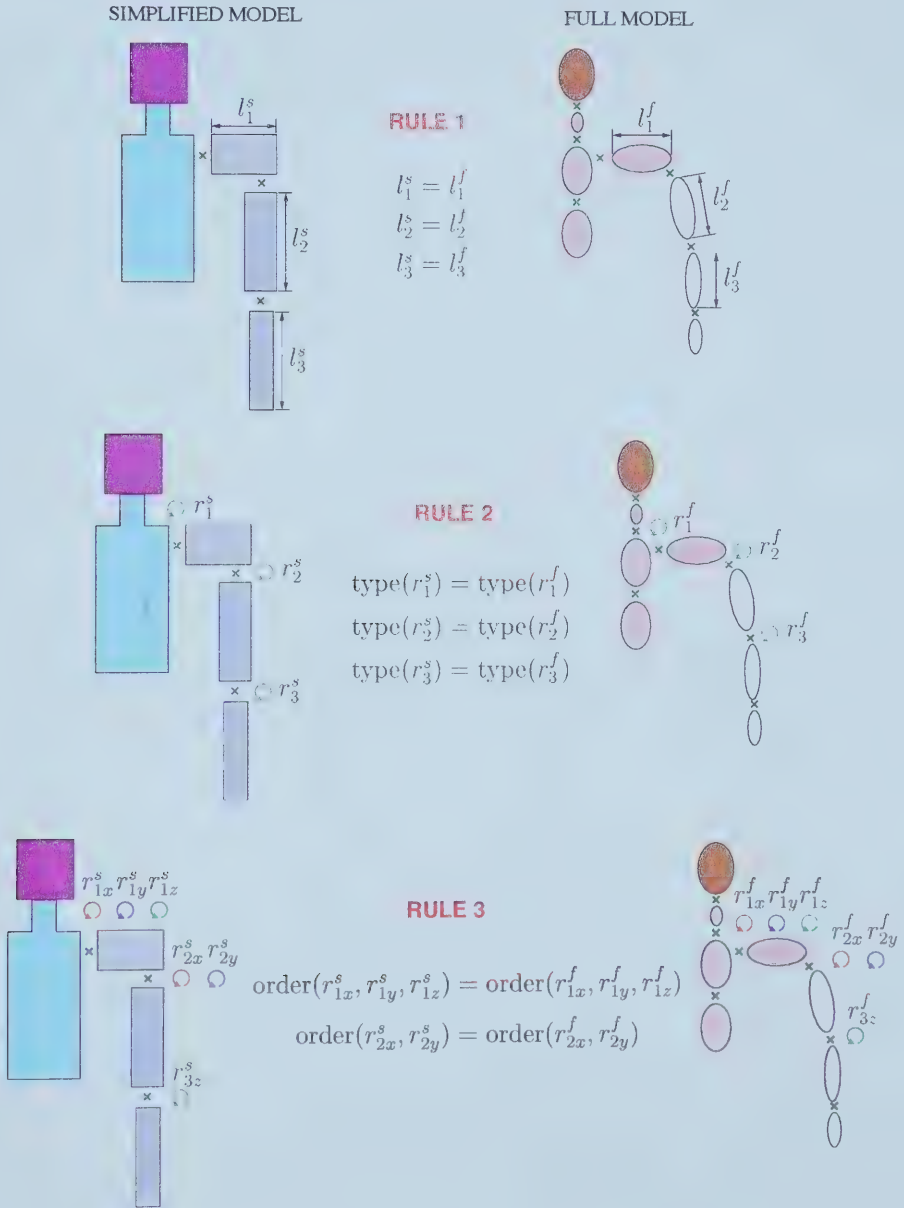


Figure 4.16: The three DDS skeleton matching rules

of the **Human** model containing the original motion. This forced us to substitute every single DOF from the **Walker** model; although, it was not the initial intention.

These rules might seem scary for the model designer; but when they are grasped, they become very intuitive and designing the proper pair of models for the DDS upmapping becomes a piece of cake.

Chapter 5

Conclusion

We conclude this thesis with a review of objectives for our research, a review of our research results, a proposal for future research directions and a view on a future of computer games, involving human and/or animal motion. Although we were rather ambitious in our initial goals, we managed to succeed on at least two of three objectives. The goal of making motion retargeting (near) real time turned out to be a tough task. Two other goals, motion compression and motion construction with submodels were easier to achieve. In the following review of research results we present our main contribution - the method for storing motion in compressed form and applying it to the full character model in real time. Next two sections are the description of the motion construction from submodels and discussion, dedicated to the integration of Popović's system into the optical mocap software pipeline. We conclude this chapter with suggestions for future research on Popović's motion transformation system and discussion about the realism in 3D computer games. We look into the future of computer games and predict how real time motion retargeting will change them.

5.1 Review of objectives

At the very beginning of our research, we set several goals of a different priority. Goal number one was to make Popović's motion transformation system suitable for the video games domain. This involved a primary objective - to push the system to its limits of speed and to see if it was possible to retarget motion with it in real time. Then there was a secondary objective - to see how well this system would fit in the mocap studio, how much animators could do with it, how useful it could be for altering captured motion. Yet another objective was to see if we could exploit the unique feature of Popović's system - its ability to map motion between models of different complexity. It was an alluring idea to experiment with motion simplification and its influence on the motion quality in the game. We made progress on all research directions, with varying degree of success. We solved the motion compression/upmapping problem, but our attempts to achieve fast motion retargeting produced rather modest results.

5.1.1 Near real time motion retargeting

As we stated in section 1.1, real time motion retargeting is a much needed feature these days, among the 3D game design teams dealing with large amounts of captured or keyframed motion. Currently, animating a team of soccer players, for example, requires storing large amounts of static motion data inside the game. The inability to alter motion in real time makes life hard for the game designers. It is possible to blend motion, creating relatively smooth, consistent and long animations of the game characters. Unfortunately, the ability only to blend static motion is not enough to create a plausible illusion of the real life motion. In real life, animals and humans never repeat the same motion *exactly*. There are always slight variations in motion, caused by the nature of the muscular and nervous system in mammals. Hence, the “mechanical” animation of the 3D game does not look like a motion in nature. Even though the first several minutes of watching a game might not create a feeling of motion *artificiality*, it will inevitably appear later.

So the solution to the problem of motion artificiality is to introduce some variety in motion. This is obviously a bad idea if variety is introduced by increasing the number of motion samples in the game. It leads to increased expenses, due to the additional motion capture sessions required, and the size of the game grows as well. A much better solution would be to retarget motion during the game play. Since we got the source code for the motion retargeting system from its author, our primary goal became adjusting and tuning the performance of the system, so that it could suit the needs of the game developers. The goal could be considered achieved if we were able to retarget a short (up to 5 seconds) animation in a time interval below 2 or 3 seconds. Therefore, we were interested not only in real time retargeting, but also in *near* real time retargeting.

5.1.2 Real time motion upmapping and motion compression

The motion retargeting system we obtained was complex enough to offer multiple directions of research, so we chose an additional goal. Popović’s system employs a clever method of overcoming the instability of the numerical solver (SNOPT) on the very large problems, representing the human skeleton’s motion. To ensure SNOPT’s convergence to a reasonable solution, the system reduces the problem size by downmapping the original full-model motion to a simplified-model motion. Consequently, it has the opposite phase - motion upmapping, where the motion of the full skeleton is created by combining original full-model motion and the altered simplified-model motion. The additional goal we set was to explore the possibility of making the motion upmapping process significantly faster - real time, if possible. Directly connected to motion upmapping is the possibility of storing motion in a compressed form, as we could store simplified-model motion instead of the full-model one, thus saving space taken by the game.

Real time motion upmapping: Popović’s motion transformation system does not alter the original motion. It uses original motion mapping methods to deal with complexity of human motion. Both upmapping and downmapping phases involve SNOPT to solve per-frame optimization subproblems, hence they are relatively

slow. If we could achieve significant speedup of the motion upmapping phase, there could appear the possibility of storing motion for the simplified model, i.e. in the compressed form.

Motion compression: Because motion compression depends on availability of real time decompression technique, it was directly connected to the above-mentioned problem of real time motion upmapping. The theoretical compression ratio would depend on the details of the full and simplified models setup. Given that Popović was able to represent a human jump with a 10 DOF model, the prospect for compression ratio looked pretty bright. Consider representing the motion of an 80 DOF model with the motion of a 10 DOF model - this is a compression ratio of 8 to 1.

Our main contribution turned out to be in that area. We successfully solved the problem of real time motion upmapping, thus making it possible to store motion in the compressed form.

5.1.3 Applying motion transformation to submodels

The motion transformation system we were working with had a high quality physics simulation engine. Aside from an industrial strength nonlinear program solver package (SNOPT), it featured a large library of C++ classes for constructing all kinds of models and setting up appropriate optimization problems. Appendix C provides a list of the C++ modules in the MTS source code, including modules with classes used for the model and problem setup in the MTS. Noticing quickly the flexibility of the C++ class-based approach to building models and setting up optimization problems, we set the goal of pushing this flexibility even further by dissecting models and experimenting with motion transformation on them. Then we would combine parts of a dissected model into the whole model and examine the resulting motion. The submodel-based motion transformation would have been considered done if we had been able to perform independent motion altering operations on the submodels and then the combined new motion would look natural enough. “Natural enough” here means looking smooth and realistic enough to be considered for inclusion in a 3D game.

5.1.4 Motion transformation engine as the last stage of the mocap pipe

Overall appearance of the motion transformation system in its current research prototype state was not user friendly and required knowledge of the model/problem setup class library and the internal logic of the system’s functioning. The last objective for our research was an evaluation of the system from the point of view of a mocap operator/3D artist and a proposal of the new motion transformation system architecture, better suited for the needs of game developers. This goal does not have an evaluation criterion (the only possible one could be the feedback from game developers), but this would require implementing such a system. Hence, the objective was to build some theoretical ground on the proposed system structure and to outline the desired features of such a system.

5.2 Review of results

Here we present an overview of the contribution we made. We start with the description of what has been done to bring the numeric optimizer to real time speed. Although in general we have failed in that, we managed to achieve some progress in that direction. The most important result we have obtained is related to motion compression and real time motion upmapping. We have solved that problem and have proved it is possible to store the game motion data in the compressed format. Finally, we have explored the domain of submodel motion retargeting and have obtained reassuring results. It turned out that Popović’s motion transformation system can be used to alter the motion of the various parts of the game character model independently. The character’s model can then be assembled from the submodels in real time, yielding a new motion as a result.

5.2.1 Near real time motion retargeting

The problem with real time motion retargeting is in the difficulty of solving the nonlinear programs in real time. At the moment quite a few high quality numerical solvers exist, SNOPT being one of the best for solving large problems, but their performance is far from being real time. Making the motion retargeting code *significantly* faster (we are talking two orders of magnitude speedup here) means substituting something else for SNOPT. “Something else” means a custom numeric solver, optimized for a specific hardware architecture and exploiting features of the motion presented in a typical 3D game (low motion LOD, for example). This task was well beyond the time frame for our research. However, we were able to tweak a solver’s configuration file, `snopt.spc`, so that SNOPT was performing less iterations per a problem solving run, thus running significantly faster. In the end, possible ways to deal with the problem of SNOPT speed can be classified as follows (starting from the best down to the worst):

Custom nonlinear program solver: This is the best approach in terms of speedup achievable, and it is also the hardest one to implement. As the common numerical optimization literature suggests [40], there are always situations when a custom solver, taking advantage of the problem properties, will outperform any general-purpose solver, however high its quality.

Decreasing the number of iterations per SNOPT run: If we change the `Major iterations` line inside the `snopt.spc` SNOPT configuration file, we can significantly decrease the execution time of the SNOPT solver per problem-solving run. We tried to lower the number of iterations from 60 to 30. The speed increase is not even an order of magnitude, it is only about 60%, but the quality of solution is not noticeably worse than that of the solution obtained with the original number of iterations.

Adding an auto-stop SNOPT trigger: When looking for the optimal motion trajectories, which conform to all the constraints, SNOPT prints a detailed execution trace. The trace consists of several columns of data, one of which is of particular interest. It is called “Feasibility” and is an indicator of how well the current

quadratic program iteration (see [17] for QP definition and other related details) satisfies the nonlinear (Newtonian) constraints. During experiments with the SNOPT solver we noticed that when feasibility reaches the threshold of $1e-06$, one can safely stop SNOPT and obtain a high quality transformed motion. Sometimes feasibility reaches $1e-06$ in less than 30 iterations, so we could implement a trigger which stops SNOPT when feasibility reaches the necessary threshold. This trigger will not be activated always, as SNOPT is not guaranteed to reach necessary feasibility in less than 30 iterations, but it will definitely speed things up when the SNOPT's convergence rate is sufficiently high.

5.2.2 Real time motion upmapping and motion compression

The most interesting results were obtained in the area of the real time motion upmapping. Using the motion LOD concept (see section 1.4), we were able to devise a set of rules for building matching pairs of the full and simplified models. With these rules we could achieve motion upmapping in real time, thus enabling the storage of the motion data in compressed form.

5.2.2.1 Real time motion upmapping

When a pair of skeletons built in conformance with the DDS skeleton matching rules is used in the upmapping process, it becomes possible to substitute certain DOFs in the simplified model for the corresponding DOFs from the full model. This is what we call *direct DOF substitution* or DDS (see section 4.1.2.2). Visually, the resulting motion looks almost indistinguishable from the motion obtained by the standard upmapping algorithm with per-frame subproblems (see section 3.2.3.5). The real time upmapping technique by direct DOF substitution has its drawbacks of course. While achieving the truly real time performance, the strict set of constraints is imposed on the simplified and full skeletons. The standard method involving SNOPT is several orders of magnitude slower, but it can map motion between a much wider range of skeletons, as its matching rules are less strict than in DDS algorithm. Given that 3D games always store a set of predefined skeletons inside, the DDS algorithm seems to be a perfect fit for a 3D game.

5.2.2.2 Motion compression

With the DDS algorithm, it becomes possible to keep a set of original full-model motion and several motion deltas, each delta being a simplified-model motion. In fact, since delta motion may have significantly fewer animation channels than a full-model original motion, we can achieve savings in terms of space taken by the animation data. The compression ratio depends on the ratio between the number of full-model motions stored in the game and the number of deltas. It also depends on the relative complexity of the simplified models used for deltas. The fewer the number of DOFs is in the delta, the larger is the compression ratio.

5.2.3 Dynamic motion construction with submodels

It is possible to work with submodels in Popović's system. In other words, we can split the character's model into several parts (submodels) and alter their motion independently. The submodels could be stored with their animation channels data and then could be "snapped" together, producing a new motion. We performed several experiments with dissecting a human skeleton into two halves, the lower and the upper. We were able to alter both upper and lower torso movements independently and then construct new animations on the fly. This technique of the dynamic motion construction from submodels can be a useful addition to the motion compression concept, described above. Instead of storing two full-model animations with the same gait, but with different arms' motion, we can separate them and store one animation for the lower torso submodel and two animations for the upper torso submodel.

5.3 Future research directions

With a system of such complexity as Popović's, one can find numerous research directions to pursue. In this section we present only some of them. First, the concept of DDS is naturally applicable to the downmapping phase. With DDS downmapping and upmapping, the motion transformation system virtually turns into a single-phase system instead of being a three-phase system. That is, SNOPT is excluded from the two motion mapping phases. The other direction to follow is the implementation of the custom nonlinear program solver. This is a very large topic in itself, as nonlinear programming is an area of active research and no perfect method exists for solving nonlinear programs numerically. There are many considerations that should be taken into account before implementing the custom solver. Finally, we present a short proposal on the design of the motion transformation add-on for the optical mocap pipe and/or for the 3D artists/game designers.

5.3.1 Applying DDS concept to the downmapping stage

The set of the skeleton matching rules described in section 4.2.5 was initially designed for the upmapping phase only. But since there is some similarity between two motion mapping phases (down and up), this set of rules is applicable to motion downmapping. In this case, we would have another pair of skeletons, this time built not for upmapping but for downmapping. With the addition of DDS downmapping, the motion transformation system now has only one phase involving SNOPT - the motion transformation phase. The main feature of this kind of change (three-phase to single-phase) is not speed, although it is important too. As a single phase system, Popović's motion transformation system becomes much more user-friendly. It relieves 3D artists/mocap operators from doing the additional work of motion mapping. Now they could focus on the motion transformation itself.

5.3.2 Replacing SNOPT with the real time or near real time solver

The arsenal of the contemporary mathematical methods for solving nonlinear programs is quite impressive, but the diversity of methods to choose from has its drawbacks. Each method is best suited for the particular type of nonlinear program, and implementing the fastest possible custom solver for the in-game motion retargeting system requires extensive research in that area. Since one of the requirements for such a system would be ability to vary between computation speed and the quality of resulting motion, the approach where an operator could choose between those criteria would make sense. Certain optimization algorithms, described in [20], [55] and [23], deserve attention.

Anyway, if such a solver is done and it is indeed fast enough, it will literally open new horizons for motion transformation in games. Real time motion retargeting with all its benefits becomes a real possibility. This is the topic of primary importance for future research on this system.

5.3.3 Motion transformation system as the last stage of mocap pipe

Section 1.1 talks about the disadvantages of optical motion capture. The main problem with mocap is that all the data is static and there is no easy way to edit it while preserving the natural, vivid feel of the original motion. The motion transformation system we were working on fits into the mocap data postprocessing pipeline nicely. Combined with the DDS method for motion upmapping and downmapping (see section 5.3.1) such a system is essentially a filter put at the output end of the mocap pipeline. This filter will require the careful setup of all the initial, intermediate (i.e. simplified) and final skeletons. In fact, the initial and the final skeleton could be the same, but the simplified skeleton would definitely vary depending on the kind of motion the mocap operator or 3D animator is working with. If motion compression is not important and the simplified skeleton is complex enough, no change of the simplified model will be necessary during a motion retargeting session. When the simplified skeleton is complex enough, it can represent any type of motion the mocap operator could desire.

On figure 5.1 you can see a rough approximation of the proposed system's user interface.

Global and file controls: Global operations - loading mocap data file, saving it, starting/stopping optimizer, etc.

Constraint controls: Operations on constraints. Constraints can be of several types - angular, floor, space, etc. They set how the motion is to be changed.

Motion displays: Simultaneously render motion of both the full and the simplified models.

Motion playback controls: Standard set of play/stop/pause/rewind buttons, allowing operator to view/freeze/rewind animation, frame by frame or in fast rewind mode.

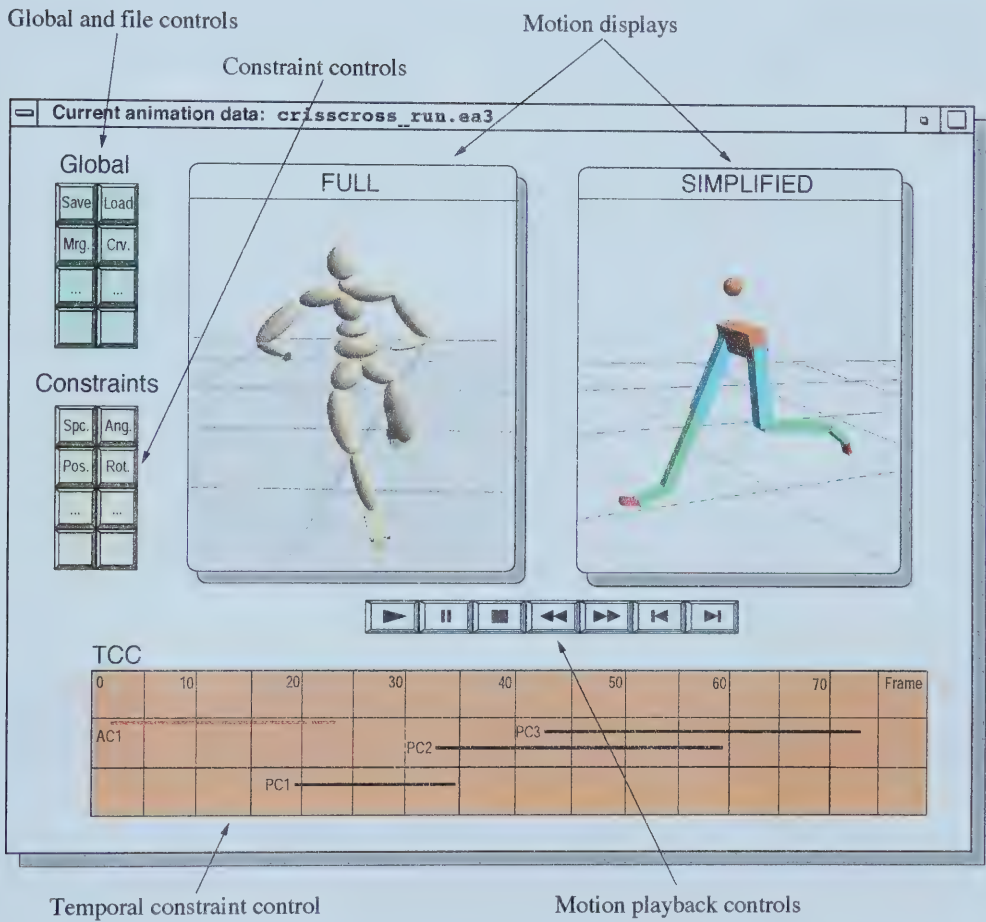


Figure 5.1: What the user interface of the motion transformation system for mocap studio might look like

Temporal constraint control: Since each constraint has its activity interval, operator must be able to control those, to change motion in the appropriate moments of time. Temporal constraint control diagram allows operator to do that. He/she can add/remove constraints as well as precisely adjust their activity intervals, with frame precision.

Architecturally, our system would contain the same core parts as those of Popović's, but the logic of their interaction would be slightly different. As you can see on figure 5.2 Popović's system has three phases, two of which we have managed to do away with. The

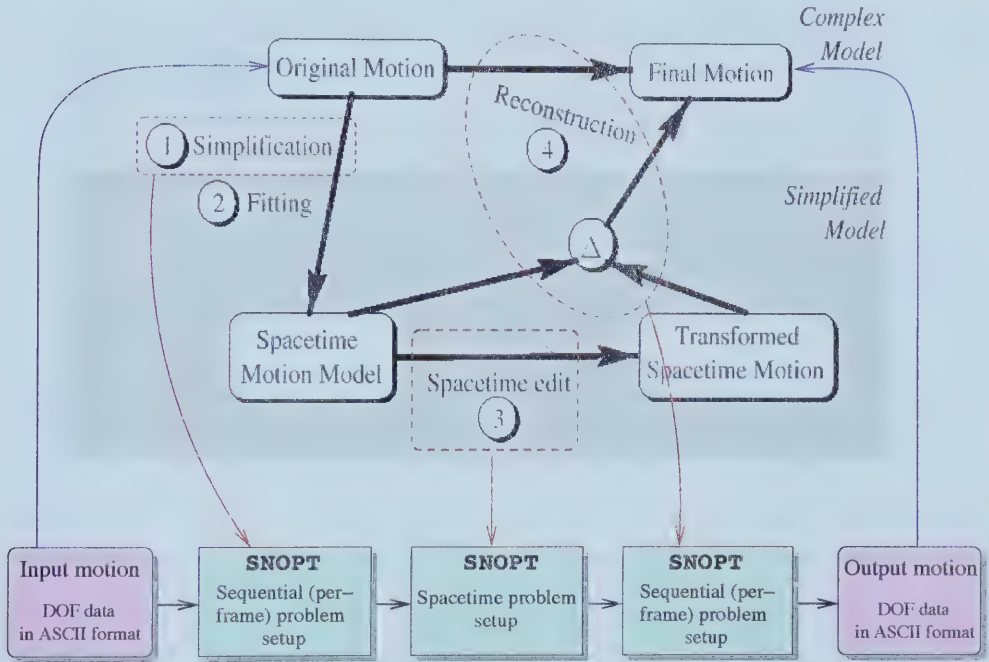


Figure 5.2: Algorithmic/architectural diagram of Popović's system

only phase left in our system is the motion transformation phase. Consequently, if you look at figure 5.3 you will notice that there is no standalone motion mapping phases in our system. To the left and to the right of the SNOPT solver module (green) you can see yellow submodules, doing DDS mapping. Thus, input motion data is loaded and *immediately* mapped onto the simplified model. After an operator edited motion constraints, he/she clicks the "Solve" button and the system performs motion transformation. After that, the transformed motion is again *immediately* mapped up, onto the full model, and is available for viewing, further editing, or saving in compressed (simplified-model) or in raw (full-model) form.

From the user's point of view, the only thing our system is able to do is motion transformation. There are no visible controls regarding motion mapping, as it happens

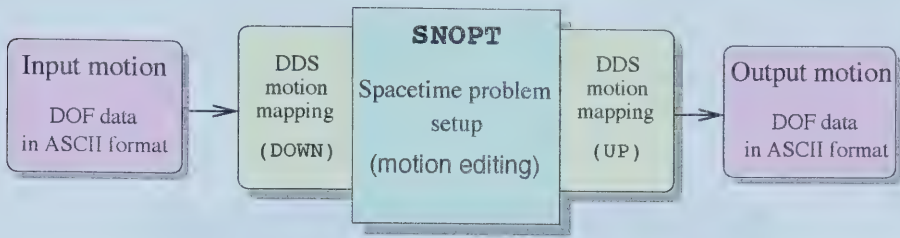


Figure 5.3: Architectural diagram of the mocap motion filter

automatically by the direct DOF substitution. To the user, system consumes original motion data, asks for constraints and other parameters regarding motion transformation, and produces transformed motion data.

5.4 What lies ahead

With computer graphics hardware and software growing and becoming more complex, the quest for maximum realism continues. In computer games, photo-realistic rendering is of a primary importance. The high quality image on the computer or TV screen creates the illusion of immersion in the game world. The more realistic and vivid is the image, the harder it becomes to distinguish between the real world and the game world. Look at figure 5.4 for the demonstration of how far realistic rendering of a human face has gone recently. Rendering quality and art are both important factors in creating this “perfect illusion”, but static images alone are not enough to accomplish this. Since motion is life, we have to create a *living* world on the screen. Hence, the challenge to the animator is the problem of realistic motion synthesis. If we look at how animation of the human characters is done in the modern 3D games, we will find either simpler keyframed animation, sometimes lacking natural feel, or captured motion, which is perfect from the point of view of the human perception system, but which is, unfortunately, unalterable. Motion transformation deals with the problem of captured motion being unalterable, thus combining better sides of two worlds.

Still, to make the physically based motion transformation usable in the games, one has to make sure the tools employed by the artists and game designers are comfortable and allow for high productivity. To design such a system, the development team has to constantly observe the artists and animators at their workplaces and interact with them, asking for a feedback during each development stage. Without the proper communication between mathematicians (developers) and animators (users) the final product most likely will be close to unusable for the animators.

The ability to introduce perturbations in motion in real time and still keep it perfectly realistic and smooth would yield major benefits for the computer game industry. Not only would the problem of monotony and artificiality in motion disappear, also a wide range of new possibilities in human character animation would arise. Adapting motion of the characters to the changed conditions in the game (limb injuries, changed weather, changed character’s health, etc.) is one possibility. Creating the game characters from scratch in

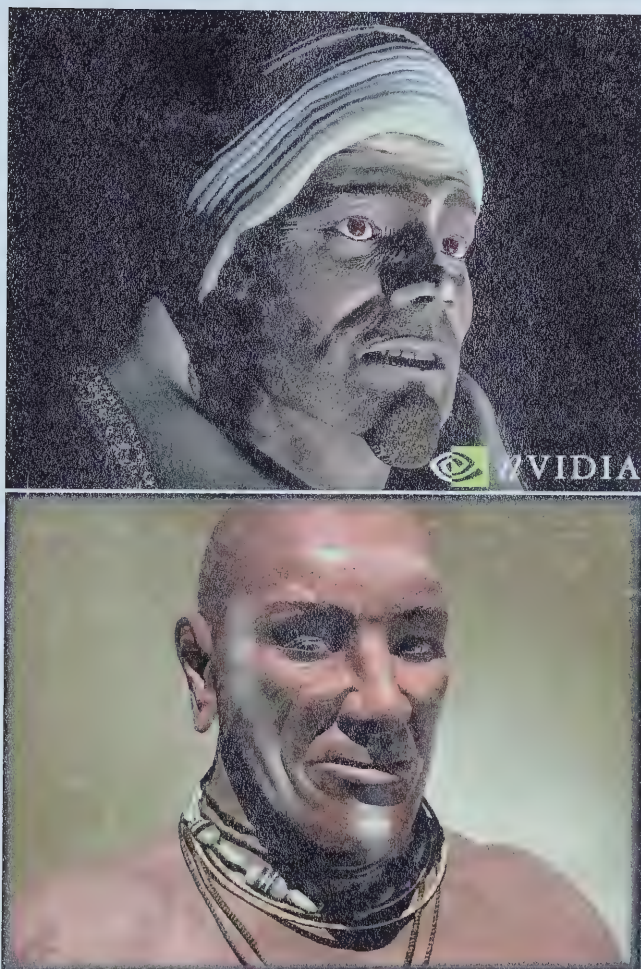


Figure 5.4: These faces are rendered, not real. Can you believe it?

the role playing games and assigning them a sort of “motion flavor” or a funny handicap is another. Do you want a left-handed dwarf with one leg shorter than the other and an asymmetrical head? No problem! Click a button and the game engine will recalculate the character’s animation data on the fly, adapting it to the changes in the character’s anatomical structure which you have just made. In short, with the real time motion retargeting the possibilities are endless. You name it!

Appendix A

Disney principles of animation

| Walt Disney's principles of animation | |
|---------------------------------------|---|
| Squash and stretch | To look realistic and convincing, animated creature cannot retain the same shape during animation. For example, if Mickey Mouse smiles, drawing a curved line across his face and animating only that line will make his face look like that of a wax doll's face. The proper way is to animate the whole face and probably the body too, stretching and squashing its parts to create proper expression. |
| Anticipation | As most of the actions in real life are preceded by some kind of anticipation motion, the same must be true for drawn actors as well. In the case of cartoon animation anticipation should be even exaggerated, to clearly show the audience what the character is going to do and leaving no chance for misinterpretation. |
| Staging | This principle is quite general and has many sides. The character should be <i>staged</i> properly, meaning that camera setup, type of shot, background and the interaction between the main character and other objects in the scene should be chosen very carefully. A part of this principle is the following rule, formulated by Walt Disney: "Work in silhouette so that everything can be seen clearly". This requirement was stipulated because all Disney animations were black and white, hence no overlapping was allowed. Later this rule turned out to be quite effective even for color animations, where overlapping was allowed. |
| continued on next page | |

| Principle | Description |
|---|---|
| Straight ahead action and pose to pose | These are two ways to animate pictures. In straight ahead action, the animator draws key frames sequentially, from the first to the last. The pose to pose approach does not impose limits on the sequence in which the animator makes frames. The key feature of pose to pose is the careful planning of mutual positions of objects in the scene and the precise definition of poses. While straight ahead action produces more spontaneous and zany animation, pose to pose is more methodical, thus featuring tight control over the character movements. |
| Follow through and overlapping action | In early Disney flicks it was discovered that final poses at the end of each scene looked excessively static. Characters appeared to be frozen for a moment between scenes. Follow through and overlapping action is a rule making such a phenomenon impossible. It demands that the animator never stop the motion of his/her character. Flesh, details of clothes and other objects in the scene should move nonstop. There was even the special term coined for animating static poses - a "moving hold" |
| Slow in and slow out | This principle imposes Newton's laws of motion on the animated character. According to it, the motion of an object cannot start and stop at once since there has to be slow in and slow out phases. In other words, "do not forget about acceleration" |
| Arcs | Inbetweening the key frames along straight line trajectories revealed unpleasant artifacts in resulting motion. It appeared too mechanical and jerky. The solution was to move character's extremities along arcs, not straight lines. This made life harder for inbetweeners, but Disney characters achieved a new level of realism in motion. |
| Secondary action | Secondary action is a shade of style put on the main motion. For example, if the cartoon character falls down, secondary action might make him open his eyes wide with amusement or fear, either after falling or while falling. This technique adds richness and naturalness to the character's motion. |
| <i>continued on next page</i> | |

| Principle | Description |
|----------------------|---|
| Timing | The number of inbetween frames between two key frames sets the speed of the action. Depending on how many inbetween frames were made, the character could stretch slowly or jerk frantically. Timing, or proper animation speed, became one of the most important concepts in cartoon animation. |
| Exaggeration | In Walt Disney's eyes exaggeration was the other side of realism. For he would not consider character's pose or action on the screen convincing, if it was not exaggerated. Making things look stronger, to the point of making them look <i>too</i> distorted, is what makes Disney animation so nice and lively. |
| Solid drawing | This is a vague principle. It demands a special style of drawing, suitable for animation, where the figure drawn appears to be made of plastic and has a "ready to be animated" appearance. One has to watch Disney cartoons to understand clearly what this concept is about. |
| Appeal | The underlying drawings in animation should be appealing to the human eye. This means a successful combination of shape, style, simplicity and grace. In fact, this is a matter of style and is very closely related to the solid drawing principle. Both solid drawing and appeal put serious demands on the animator, asking him, first of all, to be a great artist. |

Table A.1: Twelve basic animation principles, discovered by Walt Disney animators

Appendix B

Mathematical programming

The term "mathematical programming" has a familiar ring in the ears of a computer scientist because of the word "programming". However, the *mathematical* programming has very little in common with the word "programming" as it is normally understood in computer science. The only link is that mathematical programming algorithms are coded with common programming languages. For example, SNOPT, the solver used in the MTS, is a mathematical programming package implemented in Fortran 77. Mathematical programming software is called *codes* in the special literature. Obviously, because the term "programs" is already used for something else.

To illustrate the issues which the mathematical programming tries to solve we will give a mathematical program definition here. The mathematical programming problem is to find the minimum or the maximum of the *objective function*, subject to some *constraints*.

$$\begin{aligned} &\text{minimize} && F(x_1, x_2, \dots, x_n) \\ &\text{subject to} && c_i(x) = 0, i = 1, 2, \dots, p \\ & && c_i(x) \leq 0, i = p + 1, \dots, q \\ & && c_i(x) \geq 0, i = q + 1, \dots, r \end{aligned} \tag{B.1}$$

$F(x_1, x_2, \dots, x_n)$ is the multivariate objective function which is to be minimized. The variables x_i are usually constrained in some way. Otherwise, the problem is the *unconstrained* problem. The constraints are expressed as a set of p constraint functions c_1, \dots, c_p . When all the constraints are satisfied for some n -vector $x_k = (x_1, \dots, x_n)_k$ we say that x_k is a *feasible point*. Obviously, we look for the minimum only among all the feasible points, which constitute the *feasible region*.

Equation B.1 presents more a detailed and understandable definition of the general mathematical programming problem than do books on the subject. In references by McMillan [60] or by Gill et al. [40] the reader will find a stricter and shorter definition. One notable difference is that no text on the subject will present the set of constraint functions as three subsets; one for equalities and two for inequalities. Let us jump forward and draw a linear program example from any university textbook on linear programming. The common form for presenting constraints there is:

$$\begin{aligned} A \begin{pmatrix} x \\ s \end{pmatrix} &= b \\ l &\leq \begin{pmatrix} x \\ s \end{pmatrix} \leq u \end{aligned} \tag{B.2}$$

A is the constant matrix, b is a constant vector and s is a set of *slack* variables used to convert general inequality constraints to equality constraints. This form of constraint representation is less computationally expensive and is used as a *standard form* for linear programming problems [40]. Constant vectors l and u represent lower and upper bounds on the variables, including the slack ones. Each one of l_i or u_i can be a real number, a $+\infty$ or a $-\infty$. The standard form is equivalent to the verbose form in formula B.1.

B.1 Linear programming

Linear programming is the simplest case of mathematical programming. Our definition of the linear programming problem will use the standard form of constraint representation, as in B.2. For simplicity, we will omit explicit inclusion of the slack variables in the formula.

$$\begin{aligned} &\text{minimize} && c^T x \\ &\text{subject to} && \begin{cases} A \begin{pmatrix} x \\ s \end{pmatrix} = b \\ l \leq \begin{pmatrix} x \\ s \end{pmatrix} \leq u \end{cases} \end{aligned} \tag{B.3}$$

What makes a mathematical program a linear program? One condition which both the constraints and the objective function must satisfy. For a mathematical program to be a linear program, the objective function and all the constraints must be linear. If a function is linear, it can be expressed in form $a^T x = b$, where a is a constant vector and b is some scalar. Another way to show that the function is linear is to state that this function is a *linear combination* of its variables.

When the problem to solve is a linear program, it is much easier to find the minimum of the objective function. The most widely used method for solving LPs (linear programs) is a *simplex* method. For very large problems there are additional techniques. Consult any text on linear programming for details. A good introduction on both the simplex method and the techniques for solving large-scale LPs is given in a text by McMillan [60].

B.2 Nonlinear programming

Nonlinear programming is applicable to more complicated problems. The problems solved in physics, rigid body mechanics, liquid dynamics and other areas of the natural sciences are mostly of a nonlinear nature. Their formulation as nonlinear programs (NLP) yields

mathematical models closer to the natural phenomenon being modeled than any LP approximation. While linear programming is still an indispensable tool for solving economic and financial problems, NLP is quite often the only choice available to scientists to build a mathematical model of some problem met in nature.

NLP can be presented in two forms - as a *equality-constrained* problem

$$\begin{array}{ll} \text{minimize} & F(x_1, x_2, \dots, x_n) \\ \text{subject to} & c_i(x) = 0, i = 1, 2, \dots, m \end{array} \quad (\text{B.4})$$

or as a *inequality-constrained* problem

$$\begin{array}{ll} \text{minimize} & F(x_1, x_2, \dots, x_n) \\ \text{subject to} & c_i(x) \geq 0, i = 1, \dots, p \end{array} \quad (\text{B.5})$$

Of course, the most general representation of the NLP is when constraint functions include both types, equalities and inequalities. The nonlinear programs, shown in formulae B.4 and B.5, separate constraints into equalities and inequalities because it is simpler to derive optimality conditions this way [40]. We also should note that, in general, either the objective functions or the constraint functions must be nonlinear for the program to be nonlinear. Both the objective function and the constraint functions can be nonlinear, but by setting the simpler cases aside we can derive special methods for solving them. For example, if we set a restriction on the constraint functions to be linear and the objective function to involve the linear combination of variables in the power of 1 and 2 only, we will get a special form of an NLP called *quadratic program*. There are other special cases as well, each of them has special solution methods developed to take advantage of the particular program's structure.

Considering the most difficult case, when both the constraints and the objective function are nonlinear, there are many approaches to finding the solution. Methods for solving the NLP in its most general form are the iterative methods, searching for the optimum step by step. For each step, a *subproblem* connected in one way or the other to the original problem is formed and solved. The result guides selection of the subproblem to solve for the next step. Iterations continue until some predefined criterion is met; then the optimum considered to be found. Otherwise the method can determine that the problem is *infeasible*. I.e. there are no points which satisfy all the constraints. Infeasibility is just one of the many reasons the solver may stop. The objective function improvement may become too small with each step. In some cases the improvement may turn to zero, which may or may not indicate that the optimum has been found. Another very common problem for NLP is that the numerical solvers are capable of finding the local minima only. When the global objective function minimum is sought, the general methods are inapplicable unless the local minimum is the global minimum. If it is not, then it might be possible to design a custom solver which would exploit the NLP structure in order to find the global minimum.

The solver used in the MTS to transform motion is the commercial general-purpose nonlinear programming package, called SNOPT [39]. It is a large-scale NLP package written in Fortran 77 by the authors of the monograph on the numerical optimization methods [40]. The SNOPT is capable of solving both linear and nonlinear programs.

For LP, the simplex method is used and for NLP the sequential quadratic programming (SQP) method is used. SQP is an iterative method, solving a quadratic subproblem in each iteration. We present the SNOPT's formulation of the NLP below.

$$\begin{aligned} & \text{minimize} && F(x_1, x_2, \dots, x_n) \\ & \text{subject to} && l \leq \begin{pmatrix} x \\ C(x) \\ Gx \end{pmatrix} \leq u \end{aligned} \tag{B.6}$$

$C(x)$ is a vector of nonlinear constraint functions and Gx denotes a set of linear constraints. SNOPT's formulation of the NLP suggests the structure of the main function call inside the package. The function doing actual problem solving is called `snopt_`. Among its 43 parameters, `snopt_` has 6 parameters dealing with the constraint groups as they are defined in formula B.6. Parameters `nnObj` and `nnJac` are responsible for nonlinear terms in the objective function and in the constraints (Gx). Parameters `b1` and `bu` are the lower and upper bounds, represented by l and u in formula B.6. Parameter `iObj` sets the linear component of constraints and parameter `nnCon` sets the total amount of constraints in the problem, including simple bounds, linear and nonlinear constraints.

SNOPT is a very high-quality, but general purpose NLP package. Its versatility is an advantage and a weakness. While allowing its users to submit a very wide range of nonlinear programs to be solved, this package is not a real time software. It was designed with robustness in mind, not speed. To achieve a maximum possible speed, one has to exploit the problem's structure [40]. In other words, the customized solver (the solver that is tuned to the specific kind of nonlinear programs) is the best way to find a solution to the problem quickly, possibly in real time.

All the details about programming SNOPT can be found in the SNOPT user's guide [39]. Among the books on nonlinear programming, the best one is the exhaustive reference by Bazaraa et al. [17], containing a large and detailed overview of the theoretical background behind the numerical methods for solving nonlinear programs. From the solely practical point of view, the classical monograph by Gill et al. [40] will do the best. It contains an extremely detailed analysis of the various numerical techniques for solving linear and nonlinear optimization problems. Other shorter and less general references are books by Whittle [97], and Aoki [2]. Those interested in the history of NLP development may peruse the article by Kuhn [51].

Appendix C

List of C++ modules in the MTS

| MTS modules | |
|-------------------------------|--|
| <code>build.cc</code> | Definitions of the default problem properties. This module is useful when the system is launched but no input files are provided. It defines certain problem properties, such as range and type; plus assorted internal system parameters, mainly connected with animation playback. This module also contains an important method <code>Character::create()</code> which creates a model to work on depending on the input data. |
| <code>character.cc</code> | Methods of class <code>Character</code> . The internal representation of the model whose motion is being transformed is inside this class. It contains functions for adding constraints and <i>ports</i> to the character, as well as math callbacks used by SNOPT during optimization. The method <code>Character::loadActiveDofs()</code> stands aside. It is responsible for loading initial character DOFs from the input files. |
| <code>constr.cc</code> | All the methods of the descendants of class <code>Constr</code> . This class contains a semantic common for all constraints in the system. Descendants of this class include <code>ConstrFunc1Dim</code> (scalar non-linear constraint function), <code>ConstrPort</code> (constraint linking a <i>port</i> to some triple-valued function) and many others. Every single constraint that can be defined in the MTS should be a <code>Constr</code> 's descendant. |
| <code>constr-coef.cc</code> | Methods of the descendants of class <code>ConstrCoef</code> . This class implements common functionality of all the coefficient constraints in the system. Coefficient constraints are a special type of constraints where bounds have to be applied to the DOF coefficients. This may be needed, for example, when certain DOFs are set to be fixed at the beginning and/or the end of the problem range (see section 3.2.1.4 for additional information about DOF fixation available via the system's UI). |
| <i>continued on next page</i> | |

| Module | Description |
|-------------------------------|---|
| dof.cc | Definitions of all the methods for classes containing DOF functionality. Each DOF in the system is represented by an instance of class derived from class <code>Dof</code> . For example, class <code>BsplineDof</code> is a DOF with B-spline basis and class <code>CyclicBsplineDof</code> is a cyclic DOF with B-spline basis. Cyclic DOFs are very convenient for representing cyclic types of motion, such as walking and running. |
| func.cc | All the functions inside the MTS are defined through the class <code>Func</code> and its descendants. In fact, module <code>func.cc</code> contains a fairly versatile library of classes which can be used both in the objective function and in the constraint functions. Knowledge of the classes inside that module is crucial for the user's ability to set up motion transformation problem. |
| glrender.cc | OpenGL rendering methods for all classes able to draw their instance are gathered here. All the 3D character model classes, such as <code>Node</code> , <code>Character</code> , <code>PrimSphere</code> and <code>PrimCube</code> plus certain auxiliary problem-related classes (<code>MassCenter</code> , <code>PortForce</code> , <code>PortMuscle</code> and others) have a virtual method <code>glRender()</code> which draws the class instance inside the scene. Separating all the <code>glRender()</code> methods in the standalone module makes porting from OpenGL to other graphics interfaces easier. |
| graph.cc | A set of functions manipulating 2D graphs, and all the UI callbacks for the 2D graphs window (see section 3.2.1.5). The motion transformation system's interface has a convenient and powerful feature - the user can work with 2D DOF graphs before or after the motion has been transformed. Actually, the graphs window can display graphs not only for DOFs. Any function derived from class <code>Func</code> can be made <i>graphable</i> by calling the class's method <code>isGraphable()</code> with the parameter "1". Constraint functions (especially the Newtonian constraint functions) are primary candidates to be drawn as 2D graphs besides DOFs themselves. The module <code>graph.cc</code> contains the code framework doing most of the job of displaying and controlling the 2D graphs window. |
| hash.cc | A simple hash table implementation, with zero-terminated strings as keys and void pointers as values. For some reason, the MTS' author opted for not to use existing implementations of the hash table. |
| <i>continued on next page</i> | |

| Module | Description |
|-------------------------------|---|
| <code>interval.cc</code> | A small set of classes representing time intervals. They all are inherited from the class <code>Interval</code> . The most useful one is <code>RangeInterval</code> . It is also the most often used class. When motion data is loaded, the input file contains the time interval defining the animation timing. The correct setting of the time interval is seminal for SNOPT to work correctly. With the time interval set to be too long, SNOPT produces a totally messed up solution, because it cannot satisfy any Newtonian constraints with incorrect timing. |
| <code>io.cc</code> | Functions responsible for motion data loading and saving are located in this module. There are functions for loading and preprocessing data from the input motion file, as well as various functions for manipulating motion data already loaded. The functions whose names include “EA” were added by the author of this thesis. Finally, there are functions (also added by the author of this thesis) which are responsible for <i>direct DOF substitution</i> or DDS. See chapter 4, for the improvements and changes made in the MTS by the author of this thesis and a detailed explanation of the direct DOF substitution idea. DDS-related functions include <code>loadEA3()</code> and <code>load_ea3_dof()</code> . |
| <code>main.cc</code> | The entry point in the system (the standard entry point for every C program - function <code>main()</code>) and a set of miscellaneous functions supporting the system’s user interface. Function <code>main()</code> parses command line options passed through the <code>(argc,argv)</code> arguments and decides which input file format has been supplied by the user - an EA3 file or an ASCII text file (see section 3.2.2 for explanation of these file formats). Then it loads the input motion data, calls character and problem initialization methods to set up data structures required by SNOPT and initializes user interface. The other UI supporting functions in this module are the callbacks processing mouse and keyboard events, X11 window expose event, “UI is idle” event and a couple of other, less important, events. |
| <code>maya.cc</code> | Methods for exporting instance data to the Maya 1.5 ASCII format [1]. The MTS allows for exporting animation into the format suitable for loading into Maya, a 3D animation software. All the Maya export-related methods from the motion transformation system’s classes are gathered in this module. |
| <i>continued on next page</i> | |

| Module | Description |
|-------------------------------|---|
| <code>muscle.cc</code> | Methods of the muscle classes. The muscles of the character whose motion is being transformed are encapsulated in classes inherited from the class <code>DofMuscle</code> . The commonly used muscle class is <code>DampedDofMuscle</code> . The damped muscle semantics are explained in the MTS author's Ph.D. thesis [78]. |
| <code>node.cc</code> | The methods of class <code>Node</code> . The 3D model of the character inside the MTS is constructed from nodes. More precisely, the model is a tree of nodes. Each node has two facets, one is the math done by the node class on demand by SNOPT and the other is the OpenGL rendering of the node, translated and rotated according to its place in the model hierarchy. The major part of math is represented by the methods <code>update()</code> , <code>calculatePhysics()</code> and <code>calculateMassMatrix()</code> . These methods are indirectly called by SNOPT during a search for the problem solution and return necessary mass, location, velocity and acceleration values (and many others), given the instant in the animation time interval. The OpenGL rendering facet is contained in the method <code>glRender()</code> which is responsible for drawing the node's primitives, properly positioned and rotated in 3D space. |
| <code>objective.cc</code> | The methods of class <code>Objective</code> and of all the objective component classes. Objective components are classes inherited from the <code>ObjComponent</code> . When the user sets up a problem to solve, it is done using classes from the library in module <code>objective.cc</code> . Class <code>Objective</code> represents the objective function itself and has callbacks for SNOPT. Objective function components comprise the class library for the user to build the appropriate objective function from. For example, class <code>ObjSum2Q</code> is a component representing a sum of squares of the DOF coefficients inside. Class <code>ObjPortDistance</code> is an objective component for keeping the weighted sum of distances between ports and their desired locations in 3D space. There are nine objective components in total available for the user. |
| <i>continued on next page</i> | |

| Module | Description |
|-------------------------------|---|
| <code>pole.cc</code> | The methods of class <code>GerbilPole</code> . The “gerbil pole” is a clever way of making a 3D pointer out of a standard 2D computer mouse. By manipulating the mouse the user can precisely position the pointer in the 3D space. The gerbil pole is an indispensable tool for the current (very experimental and raw) user interface, as it prints the current 3D pointer coordinates after each movement of the mouse affecting the pointer. Since the user has to set the constraints manually by editing the source code, it is impossible to work without such a pointer. The gerbil pole is the only way to find out the precise 3D coordinates of the constraint the user wants to set. Given these coordinates, it is a matter of simple typing to put them in the source code, add the necessary constraint function “glue” and recompile the MTS. |
| <code>port.cc</code> | Class <code>Port</code> and its descendant classes, like <code>PortForce</code> and <code>PortMuscle</code> have their methods here. The <i>port</i> is a triple valued function which represents some “spot” on the character or inside it and which should serve as a reference point or as a point for applying a force or a constraint. For example, when the user wants to set a 3D positional constraint on the character’s foot, he/she must attach a port to the character’s foot and then constrain <i>that port</i> (not the foot!) to some point in space. |
| <code>primitive.cc</code> | Each node has a geometric primitive, represented by class <code>Primitive</code> . This module contains methods of class <code>Primitive</code> and its descendant classes, such as <code>PrimSphere</code> and <code>PrimCube</code> . Each primitive, just like a node, has two facets - drawing itself and doing math on demand by SNOPT. Unlike the case with a node, the primitive’s math and drawing functionality is tightly coupled. The node’s primitive has a mass, a mass tensor and a transformation (called <i>xform</i> in the MTS) attached to it. Mass, and everything else except <i>xform</i> , is used only by SNOPT; but <i>xform</i> is used both by SNOPT and by the OpenGL rendering method, <code>glRender()</code> . |
| <i>continued on next page</i> | |

| Module | Description |
|-------------------------------|---|
| <code>problem.cc</code> | Stuff belonging to class <code>Problem</code> and its descendants. Class <code>Problem</code> and its two most important descendants, <code>ProblemSeq</code> and <code>ProblemSpacetime</code> are half of the glue connecting the motion transformation logic to the nonlinear program solver, SNOPT. The other half is module <code>solver.cc</code> . The classes inside module <code>problem.cc</code> have methods for creating problem instances and initializing interface with SNOPT. They also have the necessary callbacks required by SNOPT. One should note, though, that the user never touches anything in that module, as this is the static part of the problem representation framework. The user works only with components defined in modules <code>objective.cc</code> and <code>constr.cc</code> . |
| <code>solver.cc</code> | This module complements module <code>problem.cc</code> . It has all the methods of the class <code>Solver</code> and its descendants. We can treat <code>Solver</code> as a low-level “SNOPT driver” or a SNOPT interface class. While class <code>Problem</code> takes care of the high-level data structures and overall semantic integrity of the MTS’ class tree, the <code>Solver</code> is responsible for organizing interaction with SNOPT. The code inside this module is quite complicated and is full of tiny details of setting up numerous data arrays and callback pointers for SNOPT. |
| <code>svec.cc</code> | Definitions of the methods of the utility classes used for working with sparse and dense matrices and vectors of real numbers. We can join these utility classes with the set of functions and data structures in the seven C modules described above. They all are just auxiliary classes used as convenient data containers by the other classes. |
| <code>uiforms.cc</code> | Almost all the UI callbacks are stored here, the rest are inside the <code>main.cc</code> . The callbacks inside this module are called when the user interacts with UI by selecting a menu item, clicking on a pushbutton, etc. |
| <code>view.cc</code> | Contains the methods of class <code>View</code> . This is the class that is responsible for powerful camera control functions available in the MTS’ animation viewer (see section 3.2.1.1). The methods inside <code>View</code> implement the camera rotation, scrolling and zooming. |
| <i>continued on next page</i> | |

| Module | Description |
|--|--|
| <code>xform.cc</code> | One of the largest modules in the system. Contains all the methods belonging to <i>xform</i> classes, or transformation classes in the MTS. Transformations available include Scale , Translate , RotateEuler and several others. Each instance of class inherited from Xform has one or more DOF instances inside. Each DOF can represent a particular variable in that transformation. For example, Translation can have three DOFs - one for translation along X coordinate, one for translation along Y and the third one for translation along Z. The xform classes can convert data from the DOFs inside to the matrix form via method update() and this is their main purpose. The 3D rendering of the character is done by traversing the model hierarchy, calling the update method for each xform and stacking the returned matrices to get the proper primitive translation and rotation done. |
| <code>biped.cc</code> <code>creatures.cc</code> <code>hopper.cc</code> <code>human.cc</code> <code>hybrid.cc</code> <code>torso.cc</code> | Six modules having one common feature. They are the modules which the user is supposed to edit during the problem setup stage. Each of them has a model or several model definitions inside. For example, <code>hybrid.cc</code> defines model of the character named Walker and <code>human.cc</code> has several human models inside. Naturally, before the user is to set up the problem, he/she must choose which module to edit, depending on the model desired. Then inside the module the 3D model is constructed, the objective is set and the constraints are attached to the proper ports. Once the module has been changed as needed and the model with its problem and constraints has been set up, the MTS can be compiled and run. |

Table C.1: Description of all the 31 module in the MTS source code

Appendix D

List of animation clips online

| EAHuman motion | |
|--------------------------------------|--|
| Original motion captured run | |
| Clip 1 | View from the back. The still model with V-like legs is the Biped. |
| Clip 2 | View from the front. |
| Clip 3 | Before upmapping to criss-cross run. You can see upmapping handles in this clip. |
| SNOPT-upmapped from the Biped model | |
| Clip 4 | Criss-cross run. View from the front. |
| Clip 5 | Criss-cross run. View from the side. |
| Clip 6 | Wide run. View from the back. |
| Clip 7 | Wide run. View from the front. |
| DDS-upmapped from the Biped model | |
| Clip 8 | Criss-cross run. View from the back. |
| Clip 9 | Criss-cross run. View from the front. |
| Clip 10 | Wide run. View from the back. |
| Clip 11 | Wide run. View from the front. |
| SNOPT-upmapped from the Walker model | |
| Clip 12 | Wide run. |
| Clip 13 | Criss-cross run. |
| Biped motion | |
| Downmapped | |
| Clip 14 | View from the back. The Biped model follows the full model, EAHuman. |
| Clip 15 | View from the front. |
| Clip 16 | View from the back. The same downmapped motion, but all the geometric primitives are rendered "wired" (not solid). |
| Clip 17 | Again "wired". View from the front. |
| Fit | |
| Clip 18 | View from the back. Now there is only Biped running, and the motion is "spacetime fit". |
| Clip 19 | View from the front. |
| <i>continued on next page</i> | |

| | |
|---|---|
| Clip 20 | View from the side. |
| Transformed | |
| Clip 21 | Criss-cross run. View from the back. |
| Clip 22 | Criss-cross run. View from the front. |
| Clip 23 | Wide run. View from the back. |
| Clip 24 | Wide run. View from the front. |
| Clip 25 | Wide run. View from the side. |
| RemapEAHuman motion | |
| DDS-upmapped from the Walker model | |
| Clip 26 | Criss-cross run. |
| Clip 27 | Wide run. |
| Full DDS combined motion | |
| Clip 28 | Combined motion, with legs performing criss-cross run and upper torso doing something remotely resembling saluting a crowd of people. |
| Clip 29 | This is the same combined motion, but rendered with some bug in the OpenGL library. |
| Clip 30 | Combined motion, with legs performing wide run, while torso does the motion which resembles a cheating husband trying to escape his angry wife armed with a big club. |
| Torso motion | |
| Clip 31 | An "Airplane" motion. The Torso spreads its arms. |
| Clip 32 | Original motion, in fact a motion of the human upper torso while running. |
| Clip 33 | Saluting motion. Waving its hands. |
| Clip 34 | Saluting motion with an OpenGL bug. |
| Walker motion | |
| Clip 35 | Original (unchanged) run, just downmapped from the EAHuman. |
| Clip 36 | Criss-cross run. |
| Clip 37 | Wide run. |

Table D.1: List of animation clips online [96]

Bibliography

- [1] Alias|Wavefront. Maya - 3D animation software. Online reference: <http://www.aliaswavefront.com>.
- [2] Masanao Aoki. *Introduction to optimization techniques. Fundamentals and applications of nonlinear programming*. The Macmillan Company, 1971.
- [3] William W. Armstrong and Mark W. Green. The dynamics of articulated rigid bodies for purposes of animation. In *Graphics Interface 85 Proceedings*, pages 407–415, 1985.
- [4] Electronic Arts. American McGee’s Alice. Computer game. Online reference: <http://alice.ea.com>.
- [5] Electronic Arts. Clive Barker’s Undying. Computer game. Online reference: <http://undying.ea.com>.
- [6] Electronic Arts. FIFA 2001. Computer game. Online reference: <http://fifa2001.ea.com>.
- [7] Electronic Arts. NHL 2001. Computer game. Online reference: <http://nhl2001.ea.com>.
- [8] Norman I. Badler, Brian A. Barsky, and David Zeltzer. *Making them move. Mechanics, control, and animation of articulated figures*. Morgan Kaufmann Publishers, 1991.
- [9] R. M. Baecker. *Interactive computer-mediated animation*. PhD thesis, Massachusetts Institute of Technology, 1969.
- [10] C.A. Balafoutis and R.V. Patel. *Dynamic analysis of robot manipulators: a Cartesian tensor approach*. Kluwer Academic Publishers, 1991.
- [11] David Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *SIGGRAPH 89 Proceedings*, pages 223–232, 1989.
- [12] David Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. In *SIGGRAPH 90 Proceedings*, pages 19–28, 1990.
- [13] David Baraff. Coping with friction for non-penetrating rigid body simulation. In *SIGGRAPH 91 Proceedings*, pages 31–40, 1991.

- [14] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *SIGGRAPH 94 Proceedings*, pages 23–34, 1994.
- [15] David Baraff and Andrew Witkin. Dynamic simulation of non-penetrating flexible bodies. In *SIGGRAPH 92 Proceedings*, pages 303–308, 1992.
- [16] Ronen Barzel and Alan H. Barr. A modeling system based on dynamic constraints. In *SIGGRAPH 88 Proceedings*, pages 179–188, 1988.
- [17] Mokhtar S. Bazaraa, Hanif D. Sherali, and C. M. Shetty. *Nonlinear programming*. John Wiley and Sons, 1993.
- [18] R. Blickhan and R.J. Full. Similarity in multilegged locomotion: bouncing like a monopode. *J. Comp. Physiol. A*, pages 509–517, 1993.
- [19] Bruce M. Blumberg and Tinsley A. Galyean. Multi-level direction of autonomous creatures for real-time virtual environments. In *SIGGRAPH 95 Proceedings*, pages 47–54, 1995.
- [20] H.G. Bock and K.-J. Plitt. A multiple shooting algorithm for direct solution of optimal control problems. In *Proceedings of the 9th IFAC World Congress, Budapest*. Pergamon Press, 1984.
- [21] James H. Bramble. *Multigrid methods*. Longman Scientific & Technical, 1993.
- [22] Matthew Brand and Aaron Hertzmann. Style machines. In *SIGGRAPH 2000 Proceedings*, pages 183–192, 2000.
- [23] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial. Second Edition*. SIAM, 2000.
- [24] Lynne Brotman and Arun Netravali. Motion interpolation by optimal control. In *SIGGRAPH 88 Proceedings*, pages 309–315, 1988.
- [25] Armin Bruderlin and Thomas W. Calvert. Goal-directed, dynamic animation of human walking. In *SIGGRAPH 89 Proceedings*, pages 233–242, 1989.
- [26] Armin Bruderlin and Lance Williams. Motion signal processing. In *SIGGRAPH 95 Proceedings*, pages 97–104, 1995.
- [27] Justine Cassell, Catherine Pelachaud, Norman Badler, Mark Steedman, Brett Achorn, Tripp Becket, Brett Douville, Scott Prevost, and Matthew Stone. Animated conversation: rule-based generation of facial expression, gesture and spoken intonation for multiple conversational agents. In *SIGGRAPH 94 Proceedings*, pages 413–420, 1994.
- [28] John E. Chadwick, David R. Haumann, and Richard E. Parent. Layered construction for deformable animated characters. In *SIGGRAPH 89 Proceedings*, pages 243–252, 1989.

- [29] Stephen Chenney and D. A. Forsyth. Sampling plausible solutions to multi-body constraint problems. In *SIGGRAPH 2000 Proceedings*, pages 219–228, 2000.
- [30] Michael F. Cohen. Interactive spacetime control for animation. In *SIGGRAPH 92 Proceedings*, 1992.
- [31] Microsoft Corporation. Xbox. Game console.
Online reference: <http://www.xbox.com>.
- [32] Sony Corporation. Sony Playstation 2. Game console.
Online reference: <http://www.scea.com/news/ps2.asp>.
- [33] Sony Corporation. Sony Playstation. Game console.
Online reference: <http://www.scea.com>.
- [34] Remedy Entertainment. Max Payne. Computer game.
Online reference: <http://www.maxpayne.com>.
- [35] Jean-Daniel Fekete, Érick Bizouarn, Éric Cournarie, Thierry Galas, and Frédéric Taillefer. Tictactoon: a paperless system for professional 2d animation. In *SIGGRAPH 95 Proceedings*, pages 79–90, 1995.
- [36] Fraunhofer Institute for Integrated Circuits IIS – Applied Electronics. MP3 audio compression software. Online reference: <http://www.iis.fhg.de/amm>.
- [37] Marie-Paule Gascuel. An implicit formulation for precise contact modeling between flexible solids. In *SIGGRAPH 93 Proceedings*, pages 313–320, 1993.
- [38] Wes D. Gehring. *Charlie Chaplin: A Bio-Bibliography*. Greenwood Press, 1983.
- [39] Philip E. Gill, Walter Murray, and Michael A. Saunders. User’s guide for SNOPT 5.3: a Fortran package for large-scale nonlinear programming.
Online reference: <http://www.sbsi-sol-optimize.com/SNOPT.htm>.
- [40] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical optimization*. Academic Press, 1981.
- [41] Jean-Paul Gourret, Nadia Magnenat Thalmann, and Daniel Thalmann. Simulation of object and human skin deformations in a grasping task. In *SIGGRAPH 89 Proceedings*, pages 21–30, 1989.
- [42] Henry Gray. *Anatomy, descriptive and surgical*. Courage Books, 1974.
- [43] Joint Photographic Experts Group. JPEG image compression software. Online reference: <http://www.jpeg.org>.
- [44] Radek Grzeszczuk and Demetri Terzopoulos. Automated learning of muscle-actuated locomotion through control abstraction. In *SIGGRAPH 95 Proceedings*, pages 63–70, 1995.

- [45] James K. Hahn. Realistic animation of rigid bodies. In *SIGGRAPH 88 Proceedings*, pages 299–308, 1988.
- [46] Brian Von Herzen, Alan H. Barr, and Harold R. Zatz. Geometric collisions for time-dependent parametric surfaces. In *SIGGRAPH 90 Proceedings*, pages 39–48, 1990.
- [47] Jessica K. Hodgins and Nancy S. Pollard. Adapting simulated behaviors for new characters. In *SIGGRAPH 97 Proceedings*, pages 153–162, 1997.
- [48] Jessica K. Hodgins, Wayne L. Wooten, David C. Brogan, and James F. O’Brien. Animating human athletics. In *SIGGRAPH 95 Proceedings*, pages 71–78, 1995.
- [49] id Software. Doom. Computer game.
Online reference: <http://www.idsoftware.com/killer/doomult.html>.
- [50] Yoshihito Koga, Koichi Kondo, James Kuffner, and Jean-Claude Latombe. Planning motions with intentions. In *SIGGRAPH 94 Proceedings*, pages 395–408, 1994.
- [51] Harold W. Kuhn. Nonlinear programming: a historical view. In *SIAM-AMS Proceedings, volume 9*, pages 1–26, 1976.
- [52] Joseph Laszlo, Michiel van de Panne, and Eugene Fiume. Interactive control for physically-based animation. In *SIGGRAPH 2000 Proceedings*, pages 201–208, 2000.
- [53] Philip Lee, Susanna Wei, Jianmin Zhao, and Norman I. Badler. Strength guided motion. In *SIGGRAPH 90 Proceedings*, pages 253–262, 1990.
- [54] Yuencheng Lee, Demetri Terzopoulos, and Keith Waters. Realistic modeling for facial animation. In *SIGGRAPH 95 Proceedings*, pages 55–62, 1995.
- [55] D.B. Leineweber. Analyse und Restrukturierung eines Verfahrens zur direkten Lösung von Optimal-Steuerungsproblemen (The Theory of MUSCOD in a Nutshell). Diploma thesis, Universität Heidelberg, 1995.
- [56] Jeffrey Lew. The Killer Bean 2: The Party. 3D animated movie. Online reference: <http://www.thekillerbean.com>.
- [57] Peter Litwinowicz and Lance Williams. Animating images with drawings. In *SIGGRAPH 94 Proceedings*, pages 409–412, 1994.
- [58] Zicheng Liu, Steven J. Gortler, and Michael F. Cohen. Hierarchical spacetime control. In *SIGGRAPH 94 Proceedings*, 1994.
- [59] Michael McKenna and David Zeltzer. Dynamic simulation of autonomous legged locomotion. In *SIGGRAPH 90 Proceedings*, pages 29–38, 1990.
- [60] Claude McMillan. *Mathematical programming*. John Wiley and Sons, 1975.

- [61] Stan Melax. Dynamic plane shifting bsp traversal. In *Graphics Interface 2000 Proceedings*, 2000. Demo software with rigid body dynamic simulation at <http://www.cs.ualberta.ca/~melax/bsp/demo.html>.
- [62] Alberto Menache. *Understanding motion capture for computer animation and video games*. Morgan Kaufmann Publishers, 2000.
- [63] Dimitri Metaxas and Demetri Terzopoulos. Dynamic deformation of solid primitives with constraints. In *SIGGRAPH 92 Proceedings*, pages 309–312, 1992.
- [64] Gavin S.P. Miller. The motion dynamics of snakes and worms. In *SIGGRAPH 88 Proceedings*, pages 169–178, 1988.
- [65] Brian Mirtich. Timewarp rigid body simulation. In *SIGGRAPH 2000 Proceedings*, pages 193–200, 2000.
- [66] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animation. In *SIGGRAPH 88 Proceedings*, pages 289–298, 1988.
- [67] Richard M. Murray, Zexiang Li, and S. Shankar Sastry. *A mathematical introduction to robotic manipulation*. CRC Press, 1994.
- [68] Thomas Ngo and Joe Marks. Spacetime constraints revisited. In *SIGGRAPH 93 proceedings*, pages 343–350, 1993.
- [69] Margareta Nordin and Victor H. Frankel. *Basic biomechanics of the musculoskeletal system. Second edition*. Lea & Febiger, 1989.
- [70] Alex Pentland and John Williams. Good vibrations: modal dynamics for graphics and animation. In *SIGGRAPH 89 Proceedings*, pages 215–222, 1989.
- [71] Cary B. Phillips and Norman I. Badler. Interactive behaviors for bipedal articulated figures. In *SIGGRAPH 91 Proceedings*, pages 359–362, 1991.
- [72] Columbia Pictures. Stuart Little. Movie.
Online reference: <http://www.stuartlittle.com>.
- [73] Universal Pictures. Babe: pig in the city. Movie.
Online reference: <http://www.babeinthecity.com>.
- [74] Walt Disney Pictures. Aladdin. Animated cartoon.
Online reference: <http://www.disney.com/Aladdin>.
- [75] Walt Disney Pictures. The Lion King. Animated cartoon.
Online reference: <http://lionking.hispeed.com>.
- [76] John C. Platt and Alan H. Barr. Constraint methods for flexible models. In *SIGGRAPH 88 Proceedings*, pages 279–288, 1988.

- [77] Jovan Popović, Steven M. Seitz, Michael Erdmann, Zoran Popović, and Andrew Witkin. Interactive manipulation of rigid body simulations. In *SIGGRAPH 2000 Proceedings*, pages 209–217, 2000.
- [78] Zoran Popović. *Motion transformation by physically based spacetime optimization*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1999.
- [79] Zoran Popović and Andrew Witkin. Physically based motion transformation. In *SIGGRAPH 99 Proceedings*, 1999.
- [80] Marc H. Raibert and Jessica K. Hodgins. Animation of dynamic legged locomotion. In *SIGGRAPH 91 Proceedings*, pages 349–358, 1991.
- [81] W. T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91–108, April 1983. Held in USA.
- [82] Hans Rijkema and Michael Girard. Computer animation of knowledge-based human grasping. In *SIGGRAPH 91 Proceedings*, pages 339–348, 1991.
- [83] Charles Rose, Brian Guenter, Bobby Bodenheimer, and Michael Cohen. Efficient generation of motion transitions using spacetime constraints. In *SIGGRAPH 96 Proceedings*, pages 147–154, 1996.
- [84] Jean-Paul C. Samson. Augmenting animated quadrupedal gaits via motion warping. Master’s thesis, Department of Computing Science, University of Alberta, 1999.
- [85] Ferdi Scheepers, Richard E. Parent, Wayne E. Carlson, and Stephen F. May. Anatomy-based modeling of the human musculature. In *SIGGRAPH 97 Proceedings*, pages 163–172, 1997.
- [86] Karl Sims. Evolving virtual creatures. In *SIGGRAPH 94 Proceedings*, pages 15–22, 1994.
- [87] John M. Snyder, Adam R. Woodbury, Kurt Fleischer, Bena Currin, and Alan H. Barr. Interval methods for multi-point collisions between time-dependent curved surfaces. In *SIGGRAPH 93 Proceedings*, pages 321–334, 1993.
- [88] Parallax Software. Descent. Computer game.
Online reference: <http://www.pxsoftware.com/prod01.html>.
- [89] Demetri Terzopoulos and Kurt Fleischer. Modeling inelastic deformation: viscoelasticity, plasticity, fracture. In *SIGGRAPH 88 Proceedings*, pages 269–278, 1988.
- [90] Frank Thomas and Ollie Johnston. *The illusion of life: Disney animation*. Hyperion, 1995.
- [91] Xiaoyuan Tu and Demetri Terzopoulos. Artificial fishes: physics, locomotion, perception, behavior. In *SIGGRAPH 94 Proceedings*, pages 43–50, 1994.

- [92] Munetoshi Unuma, Ken Anjyo, and Ryoza Takeuchi. Fourier principles for emotion-based human figure animation. In *SIGGRAPH 95 Proceedings*, pages 91–96, 1995.
- [93] Michiel van de Panne and Eugene Fiume. Sensor-actuator networks. In *SIGGRAPH 93 Proceedings*, pages 335–342, 1993.
- [94] Michiel van de Panne, Eugene Fiume, and Zvonko Vranesic. Reusable motion synthesis using state-space controllers. In *SIGGRAPH 90 Proceedings*, pages 225–234, 1990.
- [95] John Vince. *3D computer animation*. Addison-Wesley, 1992.
- [96] Vadym Voznyuk. RTMU animation clips online.
Online reference: <http://www.cs.ualberta.ca/~vadym/thesis/video.html>.
- [97] Peter Whittle. *Optimization under constraints. Theory and applications of nonlinear programming*. John Wiley and Sons, 1971.
- [98] Jane Wilhelms and Brian A. Barsky. Using dynamic analysis for the animation of articulated bodies such as humans and robots. In *Graphics Interface 85 Proceedings*, pages 97–104, 1985.
- [99] Jane Wilhelms and Allen Van Gelder. Anatomically based modelling. In *SIGGRAPH 97 Proceedings*, pages 173–180, 1997.
- [100] Lance Williams. Performance-driven facial animation. In *SIGGRAPH 90 Proceedings*, pages 235–242, 1990.
- [101] David A. Winter. *Biomechanics and motor control of human movement. Second edition*. John Wiley and Sons, 1990.
- [102] Andrew Witkin, Kurt Fleischer, and Alan Barr. Energy constraints on parameterized models. In *SIGGRAPH 87 Proceedings*, pages 225–232, 1987.
- [103] Andrew Witkin and Michael Kass. Spacetime constraints. In *SIGGRAPH 88 Proceedings*, 1988.
- [104] Andrew Witkin and Zoran Popović. Motion warping. In *SIGGRAPH 95 Proceedings*, pages 17–27, 1995.
- [105] Andrew Witkin and William Welch. Fast animation and control of nonrigid structures. In *SIGGRAPH 90 Proceedings*, pages 243–252, 1990.
- [106] T.C. Zhao and Mark Overmars. XForms - a GUI toolkit for X Window Systems.
Online reference: <http://world.std.com/~xforms>.

UNIVERSITY OF ALBERTA



0 1620 15205 204

B45570